Séries temporais com Prophet

Análise e previsão de dados com Python

Casa do Código | alura

ALLAN SPADINI
VALQUÍRIA ALENCAR

Sumário

- ISBN
- Agradecimentos
- Sobre os autores
- Prefácio
- 1. Pré-processamento de séries temporais
- 2. Prophet
- 3. Aprofundando nas previsões
- 4. Sazonalidade
- 5. Outliers
- 6. Validação e métricas de performance
- 7. Hiperparâmetros
- 8. Gráficos interativos com Plotly
- 9. Modelagem de feriados e eventos especiais
- 10. Múltiplos regressores

ISBN

Impresso: 978-85-5519-314-9

Digital: 978-85-5519-312-5

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Os autores gostariam de agradecer imensamente a todos aqueles que contribuíram de alguma forma para que esse livro se tornasse realidade. Especialmente, à Karoline Penteado e ao Thiago Gonçalves, pois o Allan trabalhou com eles neste tema anteriormente. Além disso, à Editora Casa do Código por ter aceitado a proposta de publicar sobre este livro e à querida Vivian Matsui, por todas as revisões didáticas.

Valquíria Alencar

À minha mãe Enny e ao meu pai Eroildo por terem sempre me incentivado nos estudos e por terem me proporcionado uma ótima educação.

A todos aqueles que me inspiraram e ensinaram na época da minha transição de carreira para a área de *Data Science*: Guilherme Silveira, Thiago Gonçalves, Karoline Penteado e Paulo Vasconcellos. Também agradeço muito à Vanessa Cardoso, bióloga assim como eu, que me fez acreditar que era possível eu entrar no mundo dos dados.

Aos meus amigos Carolina Dias e Marivaldo Torres Junior, que me apoiaram em todos os momentos quando eu decidi que queria sair do laboratório e mudar totalmente de área. Nós nos fortalecemos e nos ajudamos muito e, atualmente, cada um está atuando em diferentes esferas da Ciência de Dados.

À Juliana Silva, com quem eu trabalhei durante uns 5 anos (se não me falha a memória). Ela era uma colega de trabalho, que se transformou em uma das maiores amizades que tenho a alegria de ter. Nós passamos por tanta coisa juntas… Momentos bons e ruins,

que nos transformaram nas mulheres que somos hoje. Obrigada por todo apoio!

À Bruna Garcia e ao Gabriel Bezerra, meus primos que são como irmãos para mim. Obrigada por sempre acreditarem no meu potencial e falarem que eu ia chegar aonde queria. Não me esqueço da Bruna dizendo que eu estava me movimentando muito e que esse movimento iria gerar os resultados que eu estava buscando.

Aos amigos que a vida me presenteou: Matheus Queiroz, Julio Nunes e Mayza da Silva. Vocês foram fundamentais em tantos momentos. Obrigada por sempre me darem forças e me incentivarem. Às vezes nós não acreditamos muito em nós mesmos e ainda bem que temos pessoas assim ao redor.

Ao Allan Spadini, meu chefe, amigo e coautor deste livro. Assim como eu, ele ama Séries Temporais e me inspirou demais nesse assunto. A ideia de escrever o livro surgiu dele, depois de eu ter desenvolvido um projeto com os dados que trabalhamos ao longo dos capítulos. Então, eu agradeço demais por ele sempre me incentivar e acreditar no meu potencial.

Allan Segovia Spadini

À minha mãe Cristina, que sempre me incentivou e mostrou como chegar mais longe.

Ao Thiago Gonçalves e à Karol Penteado, que como mencionamos, trabalharam em algum momento comigo nesse tema, e isso me ajudou muito. O Thiago foi meu chefe na Alura e as conversas que tivemos no desenvolvimento de cursos para a Alura sempre foram muito enriquecedoras. Mas mais importante do que isso eram as risadas e o ambiente leve nas reuniões do time de Data Science. Por isso, quero agradecer a todos os membros do time de Data Science

da minha época, Thiago, Karol, Igor Nascimento, ao David Neves, ao Dani Siqueira e a Paula Heloisa.

Ao Caio Souza, por ter me aprovado para produzir cursos como externo para a Alura, e ao Ricardo Bugan, por ter me contratado efetivamente e incentivado a produzir cursos sobre séries temporais. Não esquecendo do Guilherme Lima, que mencionou a biblioteca Prophet e me motivou a produzir um primeiro vídeo sobre o assunto.

À Paula Albuquerque, que me incentivou a entrar na Alura.

À Liliana Alcazar Diogo que foi minha orientadora durante muitos anos na graduação, mestrado e doutorado. Tenho saudades das conversas sobre gatos. Carrego tudo o que aprendi através da sua orientação e apoio até hoje.

À Valquíria Alencar, minha amiga e coautora neste livro. Este livro nunca teria saído da imaginação se não fosse a dedicação da Valquíria. Gostaria de agradecê-la também pela parceria em todos os projetos e ideias pouco convencionais na produção de materiais desde que nos conhecemos.

Sobre os autores

Valquíria Alencar é bióloga, Mestra e Doutora em Biotecnologia Aplicada a Recursos Naturais e Agronegócio pela Universidade de Mogi das Cruzes (UMC). Atuou desde 2008 na pesquisa acadêmica dentro da área de Microbiologia, com ênfase em Biologia Molecular e Genômica. Então, apresenta um amplo conhecimento das técnicas comumente empregadas em Bioinformática, Genômica estrutural e Genômica funcional. Foi Pesquisadora Colaboradora, no programa de Biossistemas da UFABC (Universidade Federal do ABC) no período de 2018-2021.

Sua paixão por analisar dados e obter *insights* a partir deles fizeram com que ela fizesse a migração de carreira para área de *Data Science*. Atualmente, está atuando em programação e Ciência de Dados, desenvolvendo projetos que envolvem pré-processamento e análise exploratória de dados, previsões de séries temporais, aplicação de modelos de Machine Learning e Big Data. Desde 2021 é Docente em tempo integral no Grupo Alura nos cursos de *Data Science*. É apaixonada por aprender coisas novas e resolver problemas com soluções criativas. Mais informações podem ser encontradas em seu perfil do LinkedIn: https://www.linkedin.com/in/valquiria-alencar/ e no seu currículo Lattes: http://lattes.cnpq.br/7742338443535710

Allan Segovia-Spadini é geofísico, mestre e doutor em Ciências na área de Geofísica Aplicada pela Universidade de São Paulo com período sanduíche no doutorado na Universidade de Tecnologia de Delft. Possui experiência profissional como geofísico na empresa GPR-Geoscience onde atuou nas áreas de pesquisas geológicas e geotécnicas com métodos geofísicos do tipo sísmicos, elétricos e eletromagnéticos. No mestrado, desenvolveu pesquisa relacionada ao processamento de sinais sísmicos e melhoria de resolução através de deconvolução. No doutorado, desenvolveu pesquisa voltada para a inversão de dados sísmicos. A ênfase foi na tomografia sísmica através da inversão de forma de onda completa.

Atua principalmente nos seguintes temas: métodos sísmicos, tomografia, inversão, machine learning, deep learning, métodos de otimização, deconvolução, estimativa da forma de onda, análise de velocidades, processamento de sinais e imagens. Trabalhou como instrutor na área de Ciência de Dados na Alura e atualmente é coordenador de ensino de graduação também na área de dados no Grupo Alura. Mais informações podem ser encontradas em seu perfil do LinkedIn: https://www.linkedin.com/in/allan-spadini-3561b023/ e no seu currículo Lattes: http://lattes.cnpq.br/1102733054611772

Prefácio

E aí, você já pensou em prever algo no futuro? Nós podemos prever diversas situações que são dependentes do tempo. Quando fazemos isso, estamos trabalhando com o que chamamos de Séries Temporais. Então, podemos prever, por exemplo: a quantidade de vendas de um produto em meses sucessivos, o valor das ações ou fundos imobiliários, o número de casos de uma doença ou até mesmo o consumo de eletricidade em algum local.

Existem várias ferramentas para realizar essas previsões. Neste livro vamos focar em uma biblioteca desenvolvida pelo Facebook chamada Prophet, que em português quer dizer Profeta. E por que vamos utilizar essa biblioteca? Ela normalmente produz previsões com boa qualidade e também é muito personalizável e intuitiva para pessoas que trabalham com dados, mas que não possuem experiência prévia com análise de séries temporais. É uma biblioteca muito poderosa, pois consegue modelar, treinar, testar, identificar a sazonalidade automaticamente e ainda faz a validação cruzada dos dados.

Neste livro, vamos realizar um projeto com um *dataset* que contém as temperaturas da superfície do solo de todos os países do mundo. Nós focaremos nossos estudos na previsão das temperaturas do Brasil, mas com esse aprendizado você vai entender como aplicar os conceitos de tendência, *changepoints*, sazonalidade, *outliers*, hiperparâmetros e validação-cruzada em qualquer situação que envolva uma ou mais variáveis dependentes do tempo.

Com isso, você conseguir desenvolver projetos sobre a análise de séries temporais, que podem envolver:

- Previsões na área da saúde, como a previsão do número de casos e óbitos causados por determinada doença. Com a pandemia causada pelo vírus SARS-CoV-2 diversos trabalhos envolvendo a análise de séries temporais foram realizados nesse sentido;

- Dispositivos médicos como Eletrocardiogramas (ECGs) para diagnosticar condições cardíacas registrando os pulsos elétricos que passam pelo coração ou Eletroencefalograma (EEG) usado para medir a atividade/impulsos elétricos no cérebro;
- Astronomia e Astrofísica, como a observação de eventos transitórios como supernovas para entender o mecanismo da mudança do universo com o tempo;
- Finanças, já que a maioria das decisões financeiras, de investimento e de negócios são levadas em consideração com base em mudanças futuras e previsões de demandas no domínio financeiro;
- Varejo e negócios, auxiliando as empresas na tomada de decisões, pois você consegue analisar o padrão dos dados e entender sua sazonalidade, o que é útil na previsão de possibilidades e eventos futuros e no entendimento do aumento e diminuição de vendas de determinado produto.

Esses foram somente alguns exemplos! A variedade de campos de pesquisa é imensa. Então, você vai conseguir explorar e prever muitas situações!

A linguagem Python será utilizada aqui, sem a necessidade de nenhuma instalação no seu computador, pois utilizaremos o ambiente do Google Colaboratory. Além disso, outros *datasets* serão utilizados para aprofundar ainda mais as nossas análises e o entendimento sobre como o Prophet pode ser uma ferramenta incrível. Então, vamos trabalhar com dados do mercado de ações e com um *dataset* onde podemos utilizar múltiplos regressores, ou seja, teremos mais de uma variável dependente.

Público-alvo

O público-alvo deste livro são estudantes e profissionais envolvidos em *Data Science*, independente da área ou nível de experiência.

Organização do livro

O principal objetivo deste livro é mostrar como utilizar a biblioteca Prophet para realizar previsões de séries temporais.

Passaremos por todas as etapas necessárias para realizar a previsão de uma série temporal. No **capítulo 1** vamos discutir sobre o pré-processamento de séries temporais. Então, vamos carregar os dados no Google Colaboratory e realizar análises preliminares.

Logo no **capítulo 2** já realizaremos nossa primeira previsão com o Prophet. Depois dessa previsão discutiremos como interpretar os resultados obtidos.

No **capítulo 3** nos aprofundaremos na previsão. Vamos começar separando os dados em treino e teste, para entender se os dados de testes estão conseguindo se aderir bem à nossa previsão. Além, discutiremos sobre os *changepoints*: pontos de mudança identificados automaticamente pelo Prophet, mas que podem ser definidos manualmente.

No **capítulo 4** vamos entender a diferença entre as sazonalidades aditiva e multiplicativa. Ajustaremos a flexibilidade da sazonalidade e também será possível compreender as características de uma Série de Fourier.

Para finalizar a nossa previsão faremos o tratamento dos *outliers* no **capítulo 5**. Com isso, todos os dados discrepantes serão tratados. Nesse capítulo também incluímos os intervalos de confiança para que seja possível entender quais os efeitos quando alteramos o intervalo de confiança e como isso influencia a tendência da série temporal.

Para verificar como nosso modelo está performando e avaliá-lo, utilizaremos o método de validação cruzada e vamos obter as métricas de performance do model no **capítulo 6**.

No **capítulo 7** vamos explorar os hiperparâmetros e utilizaremos uma técnica chamada *Grid Search* para obter os melhores valores de hiperparâmetros para os nossos dados. O modelo obtido será salvo e futuramente poderá ser aberto para ser atualizado, sem a necessidade de ser retreinado.

Temos um bônus no **capítulo 8** que será a obtenção de gráficos interativos com uma biblioteca chamada Plotly. Esses gráficos são muito interessantes para apresentar os resultados para o nosso público-alvo.

Já no **capítulo 9**, outro dataset será utilizado para modelagem de feriados e eventos especiais. Este dataset é sobre o preço de ações e ampliará ainda mais nossos conhecimentos sobre séries temporais.

Por fim, no **capítulo 10** vamos discutir sobre os múltiplos regressores. Até então, tínhamos trabalhado apenas com duas variáveis: uma dependente (temperatura) e outra independente (datas). Porém, existem situações em que uma variável pode depender de mais fatores. Então, outro dataset será utilizado e vamos realizar previsões adicionando mais de um regressor.

Os dados

As bases de dados utilizadas neste livro estão disponíveis no repositório do GitHub: https://github.com/allanspadini/livro-val-allan.

Capítulo 1
Pré-processamento de séries temporais

Você já deve ter ouvido falar que alguém ganhou dinheiro na bolsa de valores vendendo uma ação que havia sido comprada por um valor mais baixo em uma data anterior. Ou então que o número de vendas de um produto tem potencial de crescimento futuro, ou que o número de casos de uma determinada doença sofreu pouca alteração de acordo com sua média móvel. Todos esses tópicos têm relação e podem ser analisados levando em consideração que estamos lidando com **séries temporais**.

Neste capítulo vamos apresentar o que são as séries temporais e como podemos realizar as primeiras análises sobre elas com ferramentas de ciência de dados.

1.1 O que são séries temporais?

Série temporal é uma coleção de observações feitas sequencialmente no decorrer do tempo. Normalmente, esses dados são coletados em intervalos iguais, por exemplo, uma vez por mês, uma vez ao dia ou uma vez a cada segundo.

A análise de séries temporais é um processo que utiliza estatística e modelagem para realizar previsões que podem ser extremamente importantes na tomada de decisões estratégicas. É claro que nem sempre é uma previsão exata, pois a probabilidade das previsões pode variar muito, especialmente quando lidamos com variáveis comumente flutuantes nos dados de série temporal e temos fatores externos fora do nosso controle. No entanto, a previsão de *insights* sobre quais resultados são mais, ou menos, prováveis de ocorrer do

que outros resultados potenciais ainda é bastante útil. Basta pensar nos exemplos que acabamos de citar.

Frequentemente, quanto mais abrangentes são os nossos dados, mais precisas as previsões podem ser. Portanto, a análise de séries temporais mostra como os dados mudam ao longo do tempo, e uma boa previsão pode identificar a direção em que os dados estão mudando.

Essas análises têm uma ampla variedade de aplicações em vários setores. Quando falamos em previsões, a primeira coisa que deve surgir na mente das pessoas é a previsão do tempo e clima. Porém, é algo muito mais amplo, que envolve desde previsões na área da saúde, engenharia, finanças, varejo e negócios, até estudos ambientais, estudos sociais e muito mais.

Podemos realizar as análises que envolvem, por exemplo: as vendas de um determinado produto em meses sucessivos, a temperatura em um determinado local ao meio-dia em dias sucessivos ou o consumo de eletricidade em uma área particular para períodos sucessivos de uma hora.

Outro exemplo que podemos citar é o número de casos diários de COVID-19 em determinada região do mundo. Na figura a seguir temos um exemplo de uma série temporal, que contém o número de casos diários de COVID-19 na cidade de São Paulo, no período de março de 2020 a junho de 2021.

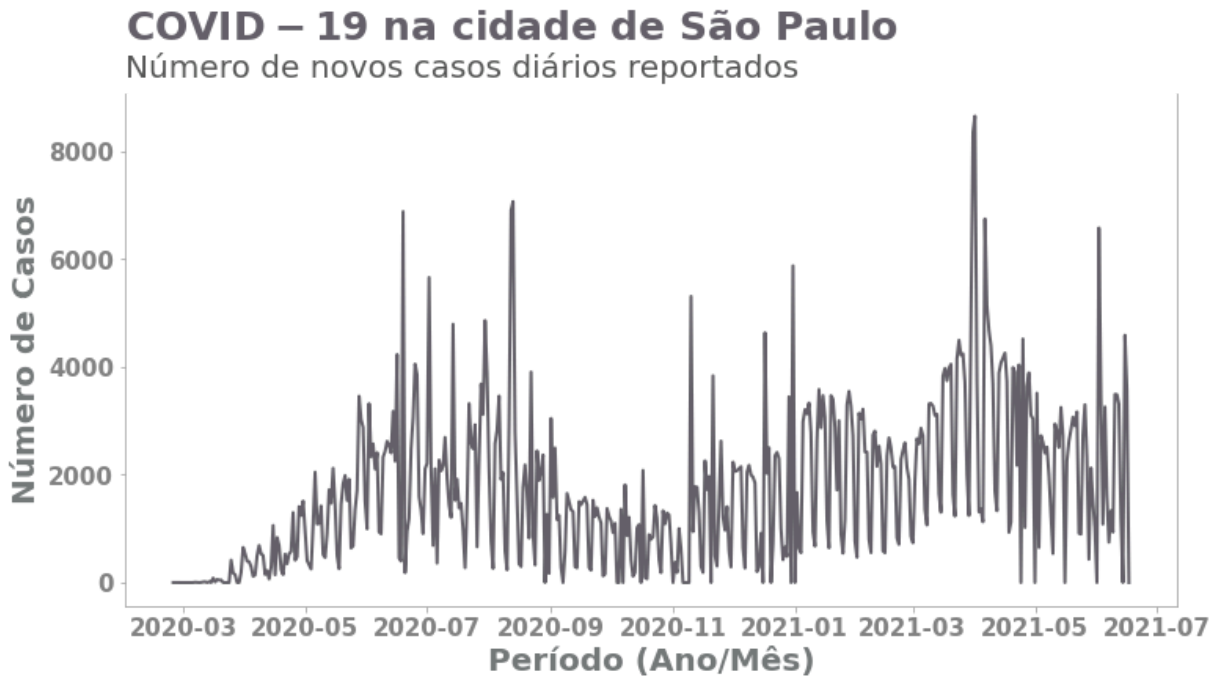

Figura 1.1: Série temporal do número de casos diários de COVID-19 na Cidade de São Paulo.

A partir dessa série temporal é possível prever a tendência do número de casos diários para os próximos meses. É difícil realizar a modelagem de séries temporais sobre COVID-19, já que temos diversos fatores externos como a testagem da população, o cumprimento da quarentena, o surgimento de novas variantes do vírus e muitos outros fatores contribuem à dificuldade de previsão. Contudo, uma análise deste tipo é muito relevante, pois pode impulsionar a implantação de estratégias de contenção e/ou avaliação de intervenções de contenção para controlar a propagação da doença.

Embora não sejam uma bola de cristal, os métodos de previsão de séries temporais podem nos ajudar a ter uma visão útil do futuro.

1.2 Análise preliminar dos dados

Para que possamos assimilar os conceitos relacionados às séries temporais, sem a necessidade de abstrações, neste livro vamos desenvolver um projeto prático. E para que você não tenha que perder tempo com a instalação de bibliotecas, vamos utilizar o **Google Colab** (https://colab.research.google.com/), que já possui a linguagem Python e todas as bibliotecas necessárias para os projetos que vamos desenvolver.

Ao abrir o Colab você vai se deparar com uma tela como esta:

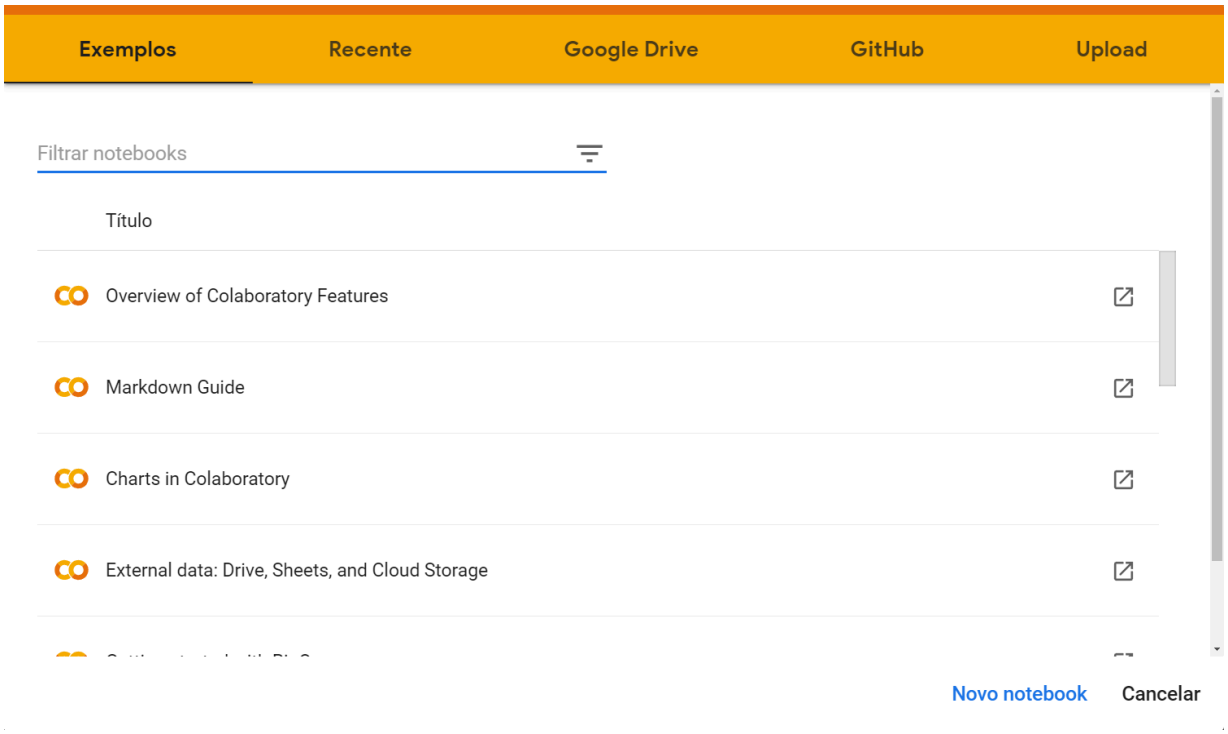

Figura 1.2: Exemplo de tela inicial quando abrimos a página do Google Colab.

Nessa tela são mostrados os arquivos mais recentes ou de exemplo para que possamos trabalhar no Colab. Chamamos os arquivos com extensão `.ipynb` de *notebooks*. Nossa intenção é criar um projeto do zero, por isso você deve clicar na opção que aparece no canto inferior direito, **Novo notebook**. Quando essa operação é executada é criado um novo notebook em branco como mostrado na figura a seguir.

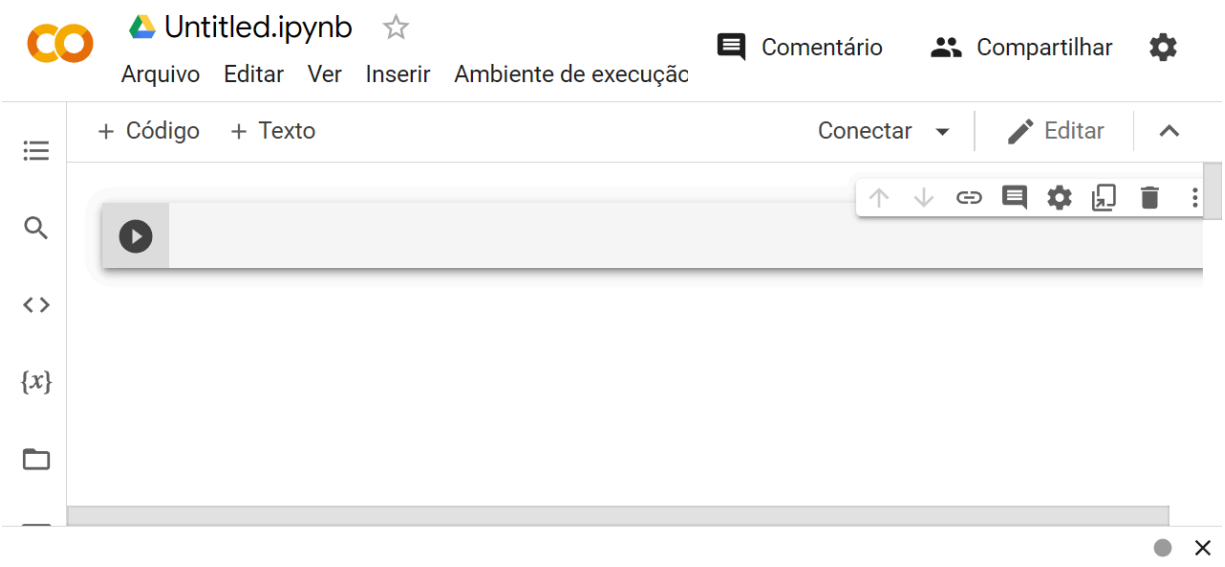

Figura 1.3: Exemplo de notebook em branco.

Nesse notebook você pode clicar no canto superior esquerdo em **Untitled** para dar um novo nome ao notebook. Nós vamos nomeá-lo como "Mudanças climáticas". Esse nome já nos dá uma dica do problema que vamos tratar.

Vamos abrir uma base de dados relacionada ao assunto de mudanças climáticas utilizando o Colab. Essa base de dados está disponível no Kaggle neste link: https://www.kaggle.com/berkeleyearth/climate-change-earth-surface-temperature-data e é proveniente de um estudo da Berkeley Earth, que combina 1,6 bilhões de relatórios de temperatura de 16 arquivos preexistentes. A Berkeley Earth é uma organização sem fins lucrativos independente, sediada em Berkeley, Califórnia, focada na análise de dados de temperatura do solo para a ciência climática.

Dos dados disponibilizados nessa página, nós selecionamos a parcela relacionada à temperatura média global do solo por país. Essa parte dos dados está disponível também no repositório do GitHub relacionado a este livro, no link: https://github.com/allanspadini/livro-val-allan/raw/main/GlobalLandTemperaturesByCountry.csv.

Voltando para o Colab, na primeira célula logo abaixo de **+Código** e **+Texto**, podemos digitar códigos em Python. Para que possamos ler o conjunto de dados, copie o endereço da base no GitHub e cole nessa primeira célula. Você deve utilizar aspas para delimitar o endereço. Além disso, vamos colocar esse endereço em uma variável de nome `url`, ficando com o seguinte resultado:

```
url='https://github.com/alura-tech/pos-datascience-time-series-forecasting/raw/main/GlobalLandTemperaturesByCountry.csv'
```

Para rodar essa célula de código, basta utilizar o comando `shift + enter` no teclado. Após a execução da célula de código, se houver uma saída, o Colab vai exibi-la logo abaixo e também criar uma célula de código nova célula de código.

A leitura do arquivo `csv` que disponibilizamos na url através de um repositório do GitHub pode ser realizada com a biblioteca **Pandas**. Para isso, podemos realizar a importação dessa biblioteca e em seguida utilizar o comando `read_csv`:

```
import pandas as pd
dados = pd.read_csv(url)
```

Nessas linhas de código realizamos a importação dando o apelido de `pd` para a biblioteca `pandas`. Em seguida, realizamos a leitura do arquivo colocando o resultado na variável de nome `dados`. A variável `dados` contém agora o que chamamos de DataFrame que neste caso vai ser uma tabela de dados. Para visualizar o nome das colunas dessa tabela podemos digitar:

```
dados.columns
```

A saída foi a seguinte:

```
Index(['dt', 'AverageTemperature', 'AverageTemperatureUncertainty',
'Country']
```

E este é o significado de cada nome de coluna:

- `dt`: ano, dia e mês;

- `AverageTemperature` : temperatura média global do solo em graus celsius;
- `AverageTemperatureUncertainty` : incerteza dos valores de temperatura coletados;
- `Country` : país.

Para facilitar nosso trabalho vamos mudar o nome dessas colunas para o português.

```
dados.columns = ['Data','Temperatura','Incerteza', 'País']
```

Para visualizar as cinco primeiras linhas desse DataFrame podemos utilizar o seguinte comando:

```
dados.head()
```

Como resultado, obtivemos uma tabela com cinco colunas e cinco linhas:

	Data	Temperatura	Incerteza	País
0	1743-11-01	4.384	2.294	Åland
1	1743-12-01	NaN	NaN	Åland
2	1744-01-01	NaN	NaN	Åland
3	1744-02-01	NaN	NaN	Åland
4	1744-03-01	NaN	NaN	Åland

Aqui já podemos observar que as temperaturas começaram a ser coletadas em 1743. Para verificar a maior data presente no DataFrame podemos digitar:

```
dados['Data'].max()
```

Como saída, você deve ter obtido a data de `2013-09-01`. Isso significa que temos em torno de 270 anos de valores de temperatura coletados no decorrer do tempo. Essa é uma quantidade grande de informações que foi coletada para cada um dos países. Que tal nos concentrarmos no Brasil como nosso exemplo de estudo? Para fazer isso, digite:

```
dados_br = dados.query('País == "Brazil"')
```

Fazendo isso, estamos colocando na variável `dados_br` os dados que na coluna `Country` possuem o valor `Brazil`.

Mas, além disso, você deve ter reparado que nas primeiras linhas nós temos diversos valores `NaN`, ou seja, valores nulos. Diversos algoritmos utilizados em análise de dados e na previsão de séries temporais podem ter problemas com dados nulos. Por isso, precisamos lidar com eles antes de iniciar o trabalho. Será que ainda restam valores nulos no conjunto de dados do Brasil?

Para verificar isso podemos escrever:

```
dados_nulos= dados_br.isnull().sum()
dados_nulos
```

O método `.isnull()` verifica se o valor dentro de uma linha é nulo e retorna `True` (Verdadeiro) caso ele seja. Como o `True` é equivalente ao número um, e `False`, ao número zero, quando utilizamos o `.sum` conseguimos somar todos os valores e obter o número total de valores nulos. A saída nesse caso é:

```
dt                                0
AverageTemperature               17
AverageTemperatureUncertainty    17
Country                           0
dtype: int64
```

Isso significa que temos 17 linhas com valores `Nan` nas colunas `AverageTemperature` e `AverageTemperatureUncertainty`.

Existem diversas formas de tratamento de valores nulos, mas para o nosso objetivo de análise inicial basta efetuar a remoção desses dados. Podemos fazer isso com o `método dropna()` da biblioteca Pandas:

```
dados_br.dropna(inplace=True)
```

Com isso, eliminamos os valores nulos. O `inplace=True` nos permitiu realizar essa operação diretamente no `dados_br` sem ter que criar outra variável.

Agora temos um conjunto de dados sem valores nulos e podemos começar a pensar no objetivo deste livro, que é realizar previsões de valores para datas futuras. Para que isso seja possível, vamos tentar entender um pouco mais sobre nossos dados utilizando a função `describe()`, que nos retorna um conjunto estatísticas descritivas:

```
dados_br['Temperatura'].describe()
```

A saída da linha de código é:

```
count    2164.000000
mean       24.699256
std         0.978787
min        21.797000
25%        24.022500
50%        24.829500
75%        25.449750
max        27.151000
Name: AverageTemperature, dtype: float64
```

Temos as seguintes informações:

- `count` : número de linhas do DataFrame (2164)
- `mean` : média das temperaturas (24.7 graus)
- `std` : desvio-padrão (0.98)
- `min` : temperatura mínima (21.8 graus)
- os valores 25%, 50% e 75% são os quartis, ou seja, são os elementos da série de dados, que a dividem em quatro partes

iguais, com o objetivo de proporcionar uma melhor ideia da dispersão do conjunto.
- `max` : temperatura máxima (27.1 graus)

Essas informações são bastante úteis quando se trata de uma previsão. Nós conseguimos, por exemplo, ver que a média de temperatura no decorrer dos anos estudados foi de aproximadamente 24,70 °C. Se quisermos "chutar" o valor da temperatura média do dia de amanhã esse valor de média seria uma boa aposta. Ou se ontem foi um dia frio, podemos supor que a temperatura de amanhã estará mais próxima da temperatura mínima observada em todo o período estudado, mas ao mesmo tempo seria pouco provável que tenhamos um valor fora do intervalo de temperaturas mínima e máxima.

Em todos esses casos estamos lidando com informações que obtivemos dos dados do passado para inferir um valor para uma data futura. De certa forma é isso que o método de previsão em séries temporais faz. Uma forma que nós mesmo podemos fazer esse processo é visualizando nossos dados através de um gráfico.

1.3 Visualização dos dados

Vamos começar tentando visualizar os dados presentes na coluna `AverageTemperature` . Para isso, vamos importar a biblioteca **Seaborn**:

```
import seaborn as sns
```

Com essa biblioteca podemos gerar um gráfico de linhas. É através de gráficos de linhas que normalmente os dados de séries temporais são expressados.

```
sns.lineplot(x='Data', y='Temperatura', data=dados_br,color='black')
```

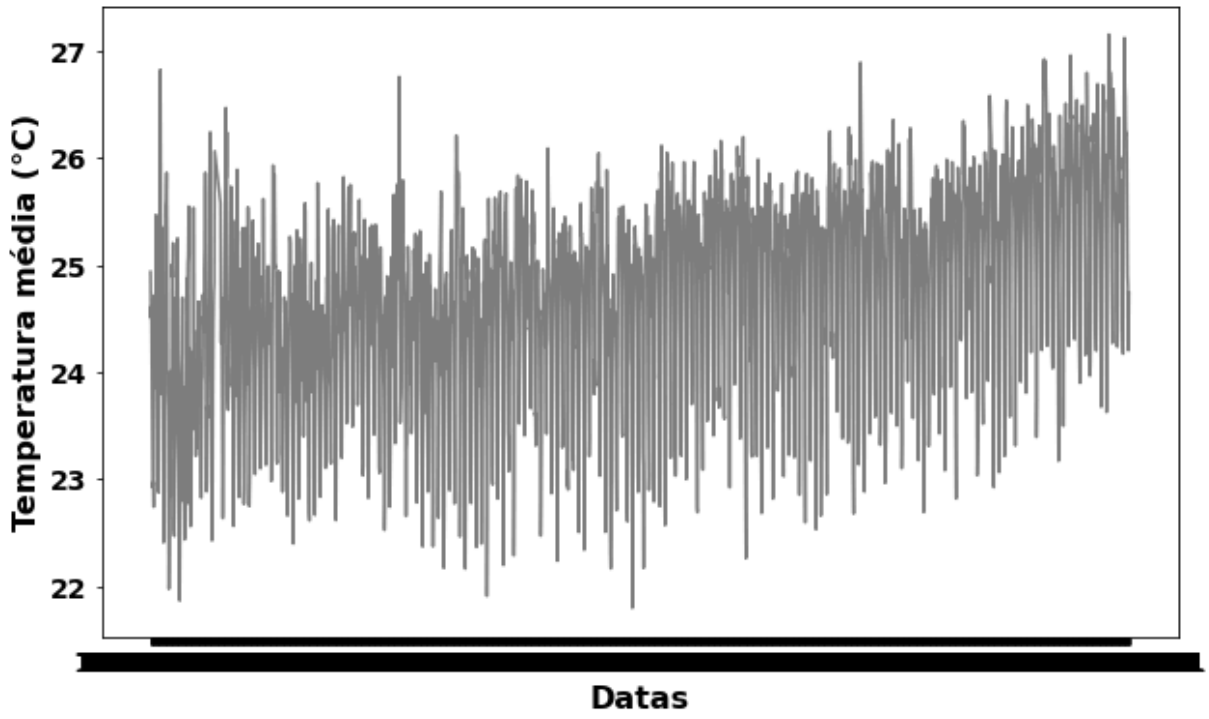

Figura 1.4: Série temporal da temperatura média pelas datas.

No eixo *y* da figura gerada estamos mostrando a Temperatura média e no eixo *x* estamos tentando mostrar as datas. Porém, no caso das datas tivemos um problema. As datas ficaram todas encavaladas no gráfico. Isso ocorre porque a Seaborn não "entendeu" que estamos trabalhando com datas e interpretou a informação de maneira incorreta na hora de gerar o gráfico. Para lidar com isso, precisamos indicar no DataFrame que estamos lidando com datas.

```
dados_br['Data'] = pd.to_datetime(dados_br['Data'], format='%Y-%m-%d')
```

Dentro do `.to_datetime` da Pandas nós adicionamos também o parâmetro `format` como um bônus que vai nos permitir selecionar os anos, meses e dias separadamente. Se agora você tentar gerar o gráfico novamente deverá obter um resultado melhor.

Mas, para melhorar ainda mais a visualização da figura, dá para ajustar o tamanho da imagem e da fonte através da biblioteca **Matplotlib**. O bloco de código completo para gerar a figura fica na forma a seguir:

```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6)) #Ajuste do tamanho da imagem
plt.rcParams['font.size'] = '14' # Ajuste do tamanho da fonte
sns.lineplot(x='Data', y='Temperatura', data=dados_br,color='black')
plt.xlabel('Datas') # Nome do eixo x
plt.ylabel('Temperatura média') # Nome do eixo y
```

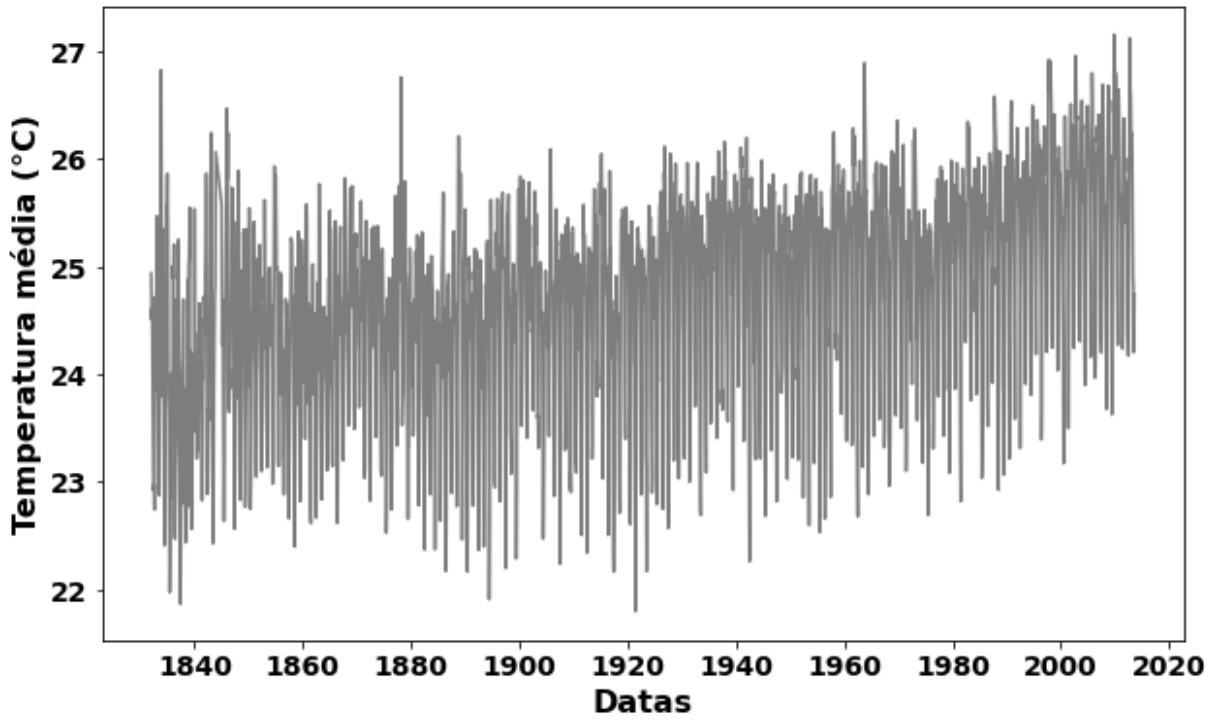

Figura 1.5: Série temporal da temperatura média pelas datas com eixo das datas corrigidas.

Agora conseguimos observar que as temperaturas variam dentro de um determinado intervalo. Porém, dentro desse intervalo a taxa de variação é bem grande. Além disso, as temperaturas aparentam ter uma tendência de aumento no decorrer dos anos. Para lidar com a variabilidade dos dados e entender melhor as variações de temperatura no decorrer de um ano, podemos olhar para média das temperaturas em um intervalo desse tipo. Por isso, vamos criar uma coluna `Mês` no nosso DataFrame:

```
dados_br["Mês"] = pd.DatetimeIndex(dados_br["dt"]).month
```

	Data	Temperatura	Incerteza	País	Mês
77110	1832-01-01	24.935	1.372	Brazil	1
77111	1832-02-01	24.505	1.953	Brazil	2
77112	1832-03-01	24.617	1.359	Brazil	3
77113	1832-04-01	23.990	2.013	Brazil	4
77114	1832-05-01	23.124	1.592	Brazil	5
...	...	...	...	...	...
79285	2013-04-01	25.464	0.253	Brazil	4
79286	2013-05-01	24.906	0.236	Brazil	5
79287	2013-06-01	24.788	0.251	Brazil	6
79288	2013-07-01	24.202	0.201	Brazil	7
79289	2013-08-01	24.736	0.247	Brazil	8

Com isso, temos a coluna com os respectivos meses de cada linha. Agora podemos agrupar os dados por mês e calcular o valor médio para cada mês. E gerar uma figura dos meses:

```
temperatura = dados_br.groupby(["Mês"]).mean()

plt.figure(figsize=(10, 6))
```

```
plt.rcParams['font.size'] = '14'
plt.bar(temperatura.index , temperatura['Temperatura'].values,color='k')
plt.xlabel('Meses do ano')
plt.ylabel('Temperatura média')
```

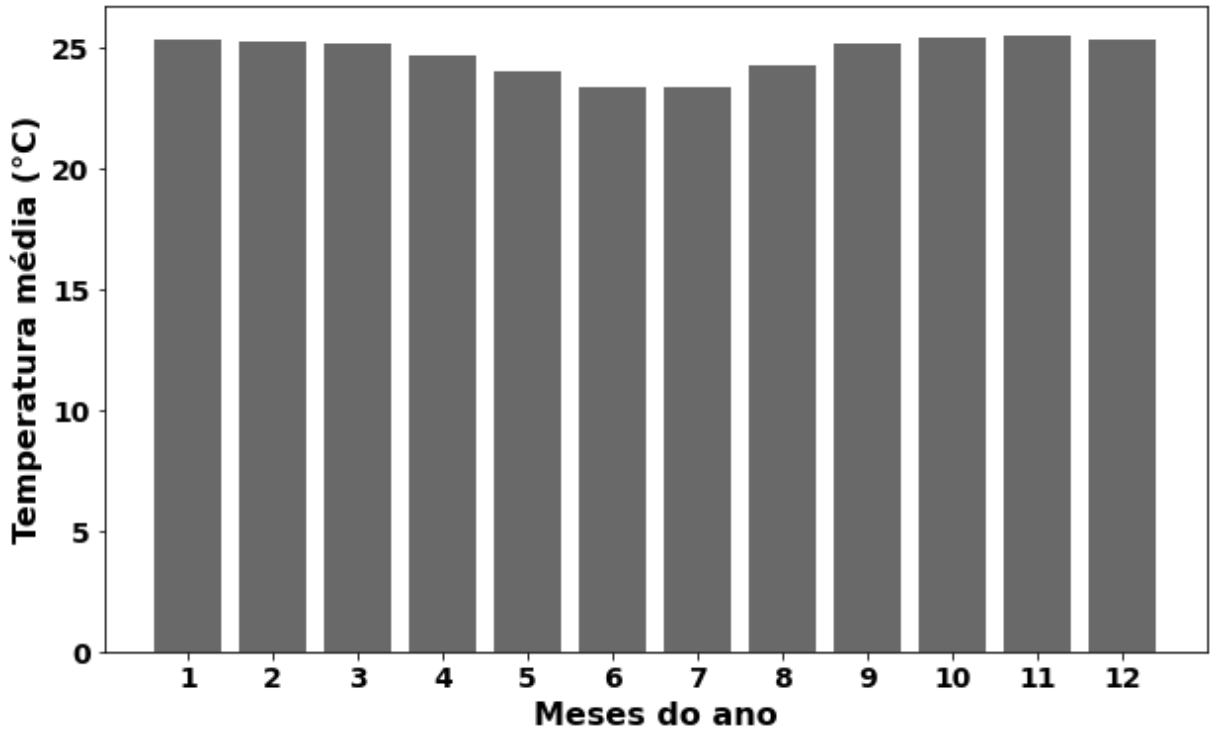

Figura 1.6: Temperatura média pelos meses do ano.

A figura da temperatura média pelos meses do ano nos permite visualizar as estações do ano, por exemplo. Que tal você começar a tentar reproduzir a análise que estamos desenvolvendo? Assim, você pode escolher um país com as estações mais bem definidas. No caso do Brasil, a variação observada não foi grande. Mesmo assim, isso já nos dá a dica de que as temperaturas são mais baixas no meio do ano. Isso pode parecer óbvio para esse conjunto de dados e país, mas é uma informação relevante caso o problema que você esteja estudando seja diferente.

Além das temperaturas médias no decorrer dos meses, uma visualização comum quando se trata de séries temporais é a da

média móvel. Vamos construir novamente a figura da série temporal com todos os anos adicionando a média móvel.

O cálculo da média móvel é feito passando uma janela sobre os dados com o método `rolling` da Pandas e posteriormente calculando a média em cada uma dessas janelas. O comando para calcular a média móvel fica na forma:

```
media_movel = dados_br['Temperatura'].rolling(30).mean()
```

No `rolling` utilizamos o valor 30 para calcular a média dos últimos 30 meses, já que a menor divisão dos dados é em meses. Agora finalmente podemos gerar o gráfico da média móvel.

```
#Ajustes da figura
plt.figure(figsize=(10, 6))
plt.rcParams['font.size'] = '14'

#Série temporal
sns.lineplot(x='Data', y='Temperatura', data=dados_br,color='gray')
plt.xlabel('Datas')
plt.ylabel('Temperatura média')
# Gera o gráfico da média móvel
sns.lineplot(x='Data',y=media_movel,data=dados_br,color='black')
plt.legend(labels=['Série temporal','Média móvel'])
```

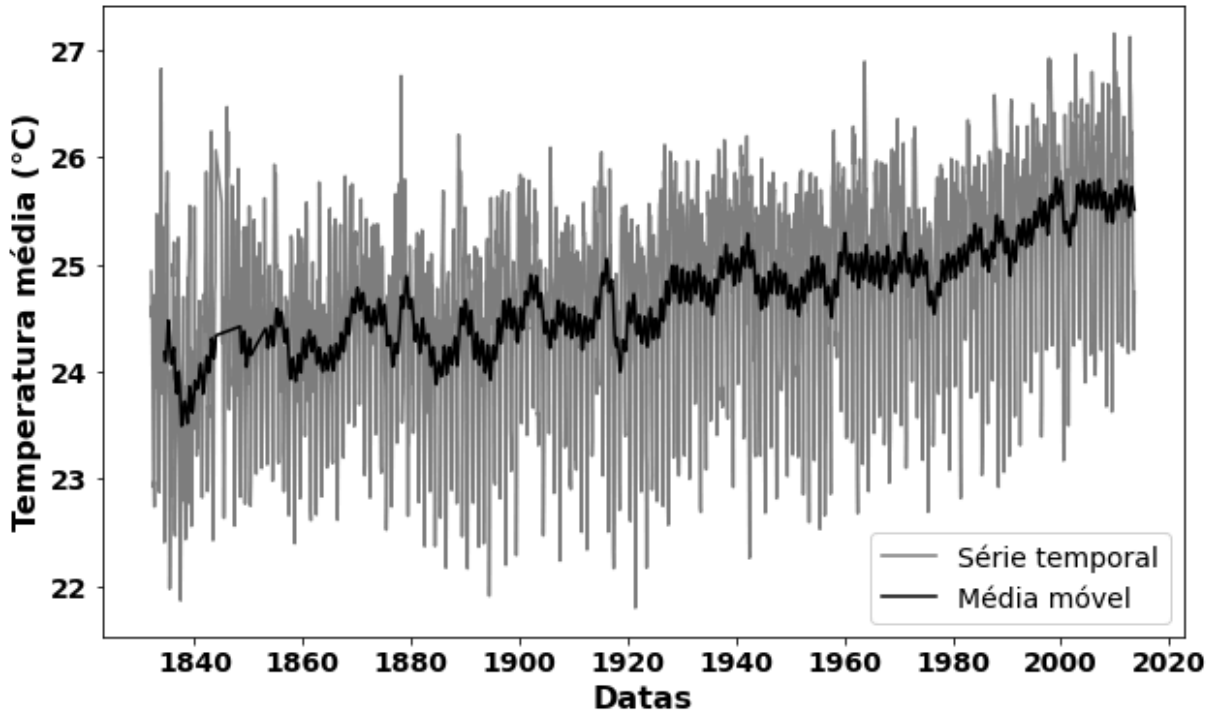

Figura 1.7: Temperatura média pelos meses do ano em cinza; Média móvel em intervalos de 30 meses em preto.

O gráfico da média móvel mostra uma variação bem menor do que a série temporal. A ideia desse tipo de gráfico é justamente essa. Dessa forma, podemos analisar a tendência geral dos dados sem nos prender às variações bruscas e pontuais que podem não se manter no decorrer do tempo.

Pensando em previsão, nós já analisamos diversas características dos dados que podem nos ajudar nesse objetivo. Se quisermos prever a média das temperaturas no mês, sabemos as temperaturas mínimas e máximas possíveis. Vimos os meses onde essas temperaturas ocorrem e também vimos que podemos utilizar a média móvel como uma maneira de visualizar a tendência dos dados. Essa tendência pode ser extrapolada para meses seguintes e assim teríamos uma previsão onde não erraríamos tanto.

Porém, você pode ter o objetivo de prever temperaturas o mais próximo possível das temperaturas que realmente serão medidas.

Será que é possível gerar uma previsão onde os valores estimados casem muito bem com os da linha da série temporal na imagem anterior para os meses seguintes? Sim, é possível! Entretanto, para isso vamos precisar conhecer uma ferramenta mais poderosa.

Capítulo 2
Prophet

A biblioteca Prophet foi desenvolvida por Sean J. Taylor e Ben Letham internamente na empresa Facebook, hoje pertencente ao grupo Meta. O lançamento da biblioteca ao público ocorreu em 2017 como um software *open source* e no artigo *Forecasting at Scale* de Taylor & Letham (2018) o funcionamento da biblioteca é descrito em detalhes. A documentação da biblioteca pode ser encontrada em: https://facebook.github.io/prophet/docs/quick_start.html.

O Prophet foi desenvolvido com o objetivo de superar alguns problemas frequentemente encontrados em outras ferramentas disponíveis para previsão de séries temporais. Logo, o Prophet foi projetado para lidar de maneira otimizada com tarefas de previsão, que podem apresentar qualquer uma destas características:

- Dados ausentes e/ou valores discrepantes;
- Dados obtidos em períodos horários, diários ou semanais, com pelo menos um ano inteiro de histórico;
- Fortes efeitos de sazonalidade ocorrendo diariamente, semanalmente e/ou anualmente;
- Feriados e outros eventos especiais, que não seguem necessariamente padrões de sazonalidade e ocorrem irregularmente;
- Mudanças significativas de tendência.

O Prophet normalmente produz previsões de alta qualidade e, além disso, também é muito personalizável e acessível para pessoas que trabalham com dados, mas que não possuem experiência prévia em análises de séries temporais. Neste capítulo você vai notar que o modelo do Prophet é bastante intuitivo.

Essencialmente, é um modelo de **regressão aditiva**. Isso significa que o modelo é simplesmente a soma de vários componentes

(opcionais), como os seguintes:

- Uma curva de tendência de crescimento linear ou logístico;
- Uma curva de sazonalidade anual;
- Uma curva de sazonalidade semanal;
- Uma curva de sazonalidade diária;
- Feriados e outros eventos especiais;
- Curvas de sazonalidade adicionais especificadas pelo usuário.

O modelo ajustado pelo Prophet é do tipo:

$$y(t) = g(t) + s(t) + h(t) + e(t)$$

Figura 2.1: y(t) = g(t) + s(t) + h(t) + e(t)

Então, a série usa o termo `g` para modelar a tendência, o `s` para as mudanças periódicas e o `h` para efeitos específicos, como feriados. O termo de erro `e` representa informações que não foram refletidas no modelo e, geralmente, é modelado como ruído distribuído normalmente.

> Para saber mais sobre modelos matemáticos utilizados em séries temporais, você pode ler o artigo do Allan, disponível em: https://www.alura.com.br/artigos/modelos-matematicos-utilizados-em-series-temporais.

No geral, o Prophet consegue alcançar resultados tão bons quanto as ferramentas de previsão mais complexas, mas com a vantagem da facilidade no uso. Inicialmente, vamos construir um modelo em poucas linhas de código. Porém, no decorrer dos próximos capítulos vamos nos aprofundar cada vez mais no modelo, adicionando mais recursos e ajustando hiperparâmetros para tentar aumentar o desempenho.

2.1 Primeiras previsões

O Prophet pode ser instalado no computador nos sistemas macOS, Windows e Linux. No entanto, em vez de fazer a instalação, vamos usar o Prophet no Google Colab. Então, para iniciar as primeiras previsões no conjunto de dados mostrado no capítulo anterior é necessário importar o Prophet a partir da biblioteca `fbprophet` :

```
from fbprophet import Prophet
```

Além dessa importação, precisamos das bibliotecas Pandas e Matplotlib, que já foram importadas no capítulo anterior.

Antes de iniciar a previsão, é necessário criar um novo DataFrame. Podemos chamá-lo de `df` :

```
df = pd.DataFrame()
```

Quando nós utilizamos o Prophet, precisamos definir duas variáveis denominadas `ds` e `y` . `ds` é a coluna dos dados que contém as datas. No capítulo anterior, essa coluna já foi formatada com a biblioteca pandas, utilizando o `pd.to_datetime` no formato esperado pelo Prophet. `y` é a coluna com os dados que vamos querer prever. Neste caso, a coluna de interesse é a coluna `Temperatura` . Então, na próxima célula vamos definir as colunas `ds` e `y` a partir de `dados_br` :

```
df['ds'] =dados_br['Data']
df['y'] = dados_br['Temperatura']
```

Analisando o DataFrame obtido, temos:

```
df
```

	ds	y
77110	1832-01-01	24.935
77111	1832-02-01	24.505

	ds	y
77112	1832-03-01	24.617
77113	1832-04-01	23.990
77114	1832-05-01	23.124
...	...	...
79285	2013-04-01	25.464
79286	2013-05-01	24.906
79287	2013-06-01	24.788
79288	2013-07-01	24.202
79289	2013-08-01	24.736

Agora podemos realizar a nossa primeira previsão. Para rodar o modelo, vamos criar a variável `modelo` e chamar o Prophet. Se você já é familiarizado com o pacote **scikit-learn** (sklearn) utilizado para *Machine Learning*, você vai notar que o Prophet foi projetado para operar de maneira semelhante. O Prophet segue aquele paradigma do sklearn de primeiro criar uma instância da classe do modelo antes de chamar o ajuste e os métodos de previsão. Então, você pode digitar:

```
modelo = Prophet()
```

Vamos fazer o ajuste do modelo, um processo conhecido como *fit*. Basta colocar o dataframe dentro do `modelo.fit` :

```
modelo.fit(df)
```

Nesse comando de ajuste único, o Prophet analisa os dados e isola a sazonalidade e tendência sem exigir que especifiquemos parâmetros adicionais. Nenhuma previsão foi feita ainda. Primeiramente, é

necessário criar um DataFrame de datas futuras e, em seguida, chamar o método de previsão, o método `make_future_dataframe`.

Entre parênteses podemos colocar algumas informações como o período, ou seja, quanto tempo de previsão queremos e a frequência: se é anual ou mensal, por exemplo. A princípio, vamos executar sem adicionar nenhuma previsão, só para checar como está o ajuste dos dados. Então, vamos digitar `periods=0`:

```
dataFramefuturo = modelo.make_future_dataframe(periods=0)
```

Por fim, criamos a variável `previsao` e chamamos o `modelo.predict`. Dentro dos parênteses adicionamos o `dataFramefuturo`:

```
previsao = modelo.predict(dataFramefuturo)
```

2.2 Interpretando o DataFrame de previsão

Vamos analisar as três primeiras linhas da previsão. Neste caso, usamos o atributo `T` para fazer uma transposição do DataFrame, melhorando a visualização dos nomes das colunas:

```
previsao.head(3).T
```

Depois de executar esse comando, você deverá ver a seguinte tabela impressa:

	0	1	2
trend	23.9627	23.9638	23.9648
yhat_lower	23.9867	23.9902	23.7351
yhat_upper	25.1014	25.101	24.9112
trend_lower	23.9627	23.9638	23.9648
trend_upper	23.9627	23.9638	23.9648

	0	1	2
additive_terms	0.599562	0.581291	0.350073
additive_terms_lower	0.599562	0.581291	0.350073
additive_terms_upper	0.599562	0.581291	0.350073
yearly	0.599562	0.581291	0.350073
yearly_lower	0.599562	0.581291	0.350073
yearly_upper	0.599562	0.581291	0.350073
multiplicative_terms	0	0	0
multiplicative_terms_lower	0	0	0
multiplicative_terms_upper	0	0	0
yhat	24.5622	24.545	24.3148

Agora vamos analisar cada uma das informações e entender o que cada uma significa!

- `ds` : data ou data/hora aos quais os valores dessa linha pertencem;
- `trend` : valor da componente de tendência sozinho;
- `yhat_lower` : limite inferior do intervalo de incerteza em torno da predição final;
- `yhat_upper` : limite superior do intervalo de incerteza em torno da predição final;
- `trend_lower` : limite inferior do intervalo de incerteza em torno da componente de tendência;
- `trend_upper` : limite superior do intervalo de incerteza em torno da componente de tendência;
- `additive_terms` : valor combinado de todas as sazonalidades aditivas;

- `additive_terms_lower` : limite inferior do intervalo de incerteza em torno das sazonalidades aditivas;
- `additive_terms_upper` : limite superior do intervalo de incerteza em torno das sazonalidades aditivas;
- `yearly` : valor da componente de sazonalidade anual;
- `yearly_lower` : limite inferior do intervalo de incerteza em torno da componente anual;
- `yearly_upper` : limite superior do intervalo de incerteza em torno da componente anual;
- `multiplicative_terms` : valor combinado de todas as sazonalidades multiplicativas;
- `multiplicative_terms_lower` : limite inferior do intervalo de incerteza em torno das sazonalidades multiplicativas;
- `multiplicative_terms_upper` : limite superior do intervalo de incerteza em torno das sazonalidades multiplicativas;
- `yhat` : valor final previsto. É uma combinação da tendência, termos multiplicativos e termos aditivos.

Em outros casos, se os dados tiverem sazonalidade semanal ou diária, as colunas para `weekly` , `weekly_lower` , `weekly_upper` , `daily` , `daily_lower` , e `daily_upper` também serão incluídas. Em capítulos posteriores vamos ter a discussão e exemplos tanto das sazonalidades aditivas/multiplicativas quanto dos intervalos de incerteza.

Para analisar o resultado da previsão, nós podemos visualizar isso de forma gráfica, com `modelo.plot` . Ao final do comando nós podemos adicionar ponto e vírgula, pois caso contrário o mesmo gráfico aparece duas vezes seguidas:

```
modelo.plot(previsao, figsize=(18,9));
plt.title('Temperaturas Médias da Superfície do Solo no Brasil ', loc =
'left', fontsize = 35)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
```

```
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

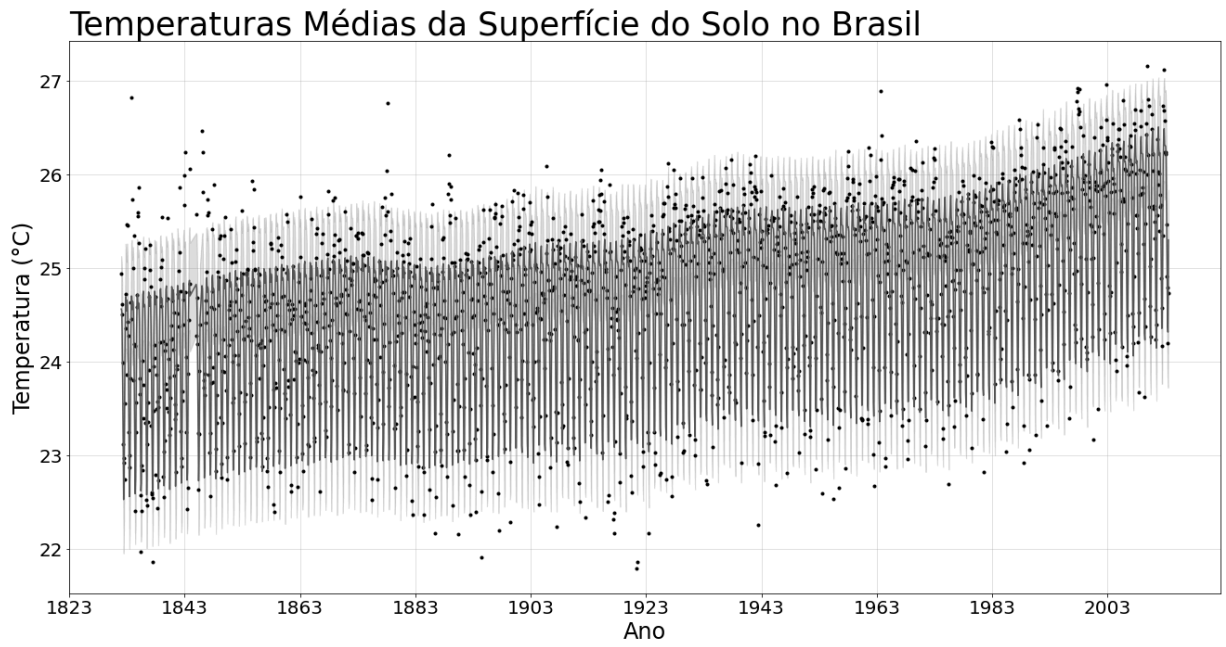

Figura 2.2: Gráfico de previsão.

Essa é a nossa previsão! No eixo *x* temos o período e no eixo *y* estão as temperaturas médias da superfície do solo. Os pontos representam os valores reais de *y*, aqueles valores de temperatura que vimos no capítulo anterior. Enquanto isso, a linha sólida em cinza escuro representa os valores de `yhat` calculados. O sombreamento mais claro ao redor a linha sólida na região prevista representa o intervalo de incerteza, limitado por `yhat_lower` e `yhat_upper`.

2.3 Compreendendo gráficos das componentes

Como o Prophet é um modelo aditivo, ele gera uma curva complexa que pode ser decomposta em suas partes constituintes: as componentes. O primeiro passo em direção ao maior controle da

previsão é entender essas componentes. Para analisar os componentes, vamos digitar:

```
modelo.plot_components(previsao, figsize=(10,6));
```

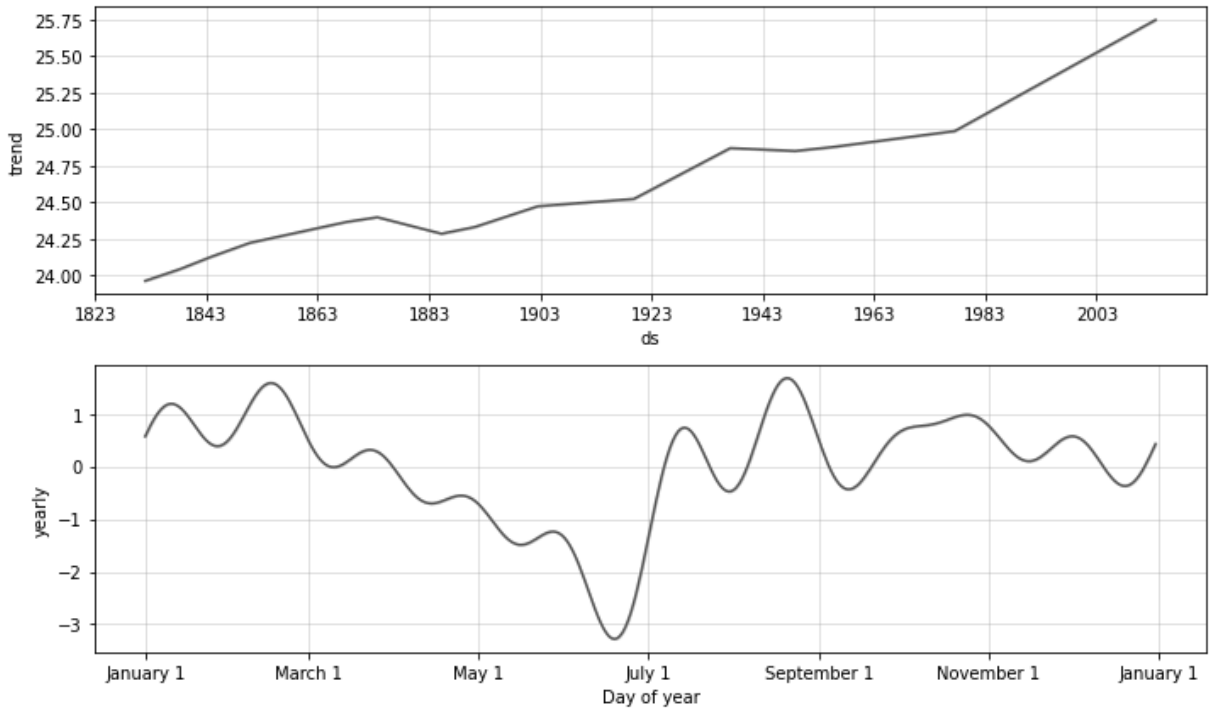

Figura 2.3: Gráficos das componentes.

Como você pode ver no gráfico de saída, o Prophet isolou duas componentes neste conjunto de dados: tendência e sazonalidade anual. Note que, na tendência, as temperaturas estão aumentando ao longo dos anos, principalmente, após 1980. A tendência aumenta constantemente, mas parece ter uma inclinação acentuada à medida que o tempo avança. Na componente anual, as temperaturas diminuem entre maio a julho e, inclusive, isso foi algo que concluímos com as análises exploratórias realizadas no capítulo anterior. Como nosso conjunto de dados tem muitos anos de dados coletados, nós vemos essa sazonalidade anual através de diversas subidas e descidas no gráfico de previsão.

2.4 Adicionando período de previsão

Agora vamos rodar o modelo, porém incluindo um período de previsão. Então, vamos digitar os comandos a seguir e usar um período de 10 anos (120 meses). A menor subdivisão dos nossos dados é em meses. Por isso, a frequência deve ser `M`:

```
modelo = Prophet()
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=120, freq='M')
previsao = modelo.predict(dataFramefuturo)
```

Vamos analisar o gráfico de previsão:

```
modelo.plot(previsao, figsize=(18,9));
plt.title('Previsão das Temperaturas Médias da Superfície do Solo no
Brasil ', loc = 'left', fontsize = 35)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

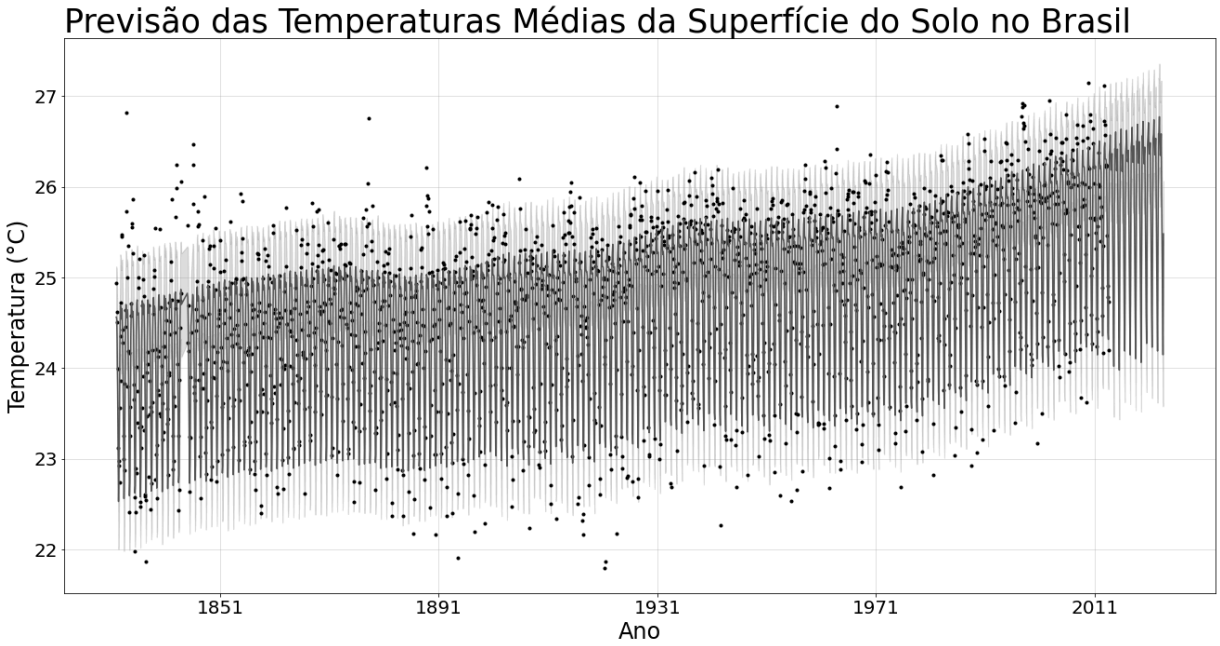

Figura 2.4: Gráfico de previsão para um período de 10 anos.

A previsão é esse período no final do gráfico onde não temos os pontos. Para entender melhor qual é a tendência para os próximos meses, que foram previstos, vamos checar as componentes:

```
modelo.plot_components(previsao, figsize=(10,8));
```

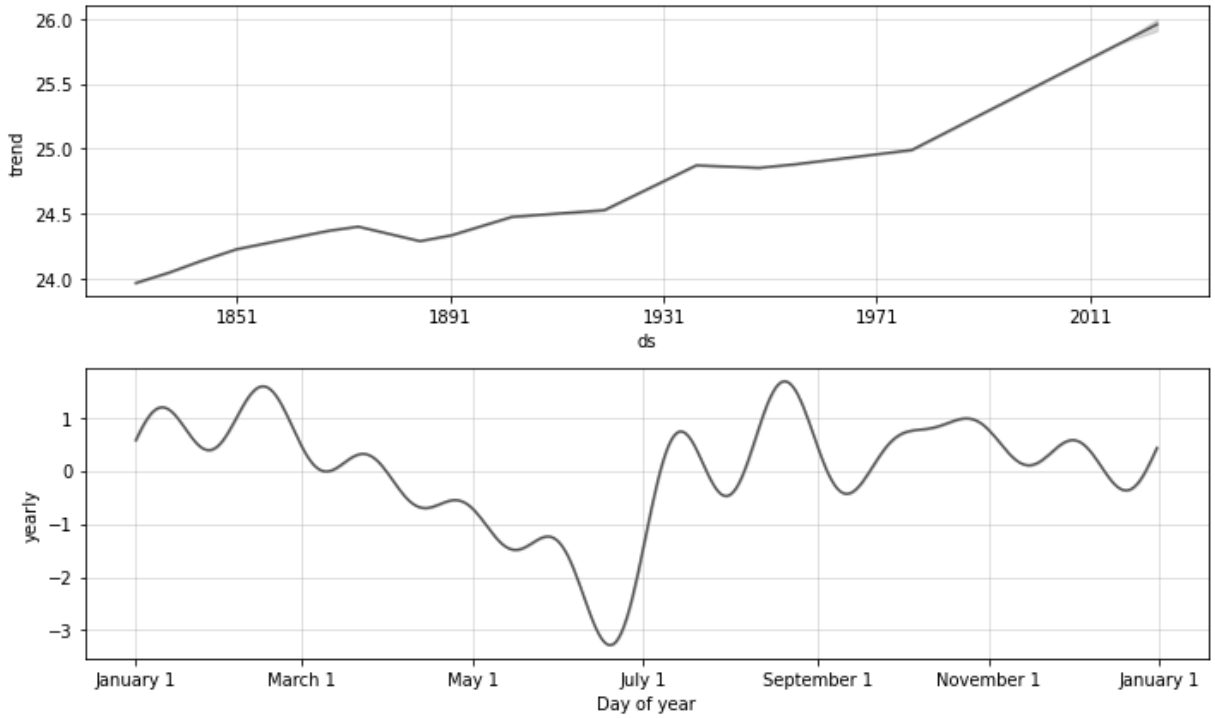

Figura 2.5: Gráfico das componentes para um período de previsão de 10 anos.

Agora a linha de tendência também mostra intervalos de incerteza nos anos previstos (aquela faixa em cinza claro). Quanto mais nos afastamos dos dados ajustados, maior é essa incerteza. Como podemos observar, há uma tendência de aumento das temperaturas médias da superfície do solo para os meses que foram previstos.

Construímos nosso primeiro modelo com o Prophet e, em apenas algumas linhas de código, conseguimos fazer a previsão para os próximos 10 anos das médias da temperatura do solo no Brasil. Também analisamos o DataFrame de previsão e foi possível entender os resultados que o Prophet entrega. Por fim, analisamos as componentes desta previsão: a tendência e a sazonalidade anual, o que ajudou a compreender o comportamento dos dados.

No entanto, temos um longo caminho pela frente. A primeira coisa que podemos nos questionar é: como saber se a nossa previsão está performando bem ou se temos um modelo que realmente faz boas previsões? No próximo capítulo vamos descobrir como fazer isso!

CAPÍTULO 3
Aprofundando nas previsões

No capítulo anterior nós fizemos as primeiras previsões com a biblioteca Prophet e também entendemos aqueles resultados de decomposição que são gerados após realizarmos as previsões. Além disso, aprendemos a analisar as componentes. Porém, faltou fazer algo muito importante! Nós também queremos saber como dados que não foram utilizados para treinar o modelo se aderem aos valores de `yhat` calculados. Quando os dados de teste apresentam uma boa aderência à previsão, isso é um indicativo de que o nosso modelo está performando bem.

Então, vamos separar os nossos dados em treino e teste. Primeiramente, um DataFrame será criado para os dados de treino:

```
df_treino = pd.DataFrame()
```

Depois disso, para definir o `ds` e o `y`, é necessário definir a quantidade de dados que serão utilizados para treinar o modelo. Nesse caso, vamos separar os 2000 primeiros dados para treino:

```
df_treino['ds'] = dados_br['dt'][:2000]
df_treino['y'] = dados_br['AverageTemperature'][:2000]
```

Em seguida, é necessário criar o DataFrame dos dados de teste. Vamos armazená-lo na variável `df_teste`:

```
df_teste = pd.DataFrame()
```

Os dados para `ds` e `y` serão do 2000 em diante:

```
df_teste['ds'] = dados_br['dt'][2000:]
df_teste['y'] = dados_br['AverageTemperature'][2000:]
```

Agora está tudo certo para treinar o modelo e realizar a previsão. O fit deve ser feito com os dados do `df_treino`. Em relação ao período, vamos prever um período de 150 meses:

```
modelo = Prophet()
modelo.fit(df_treino)
dataFramefuturo = modelo.make_future_dataframe(periods=150, freq = 'M')
previsao = modelo.predict(dataFramefuturo)
```

Após a previsão ter sido feita, é preciso plotar os dados do teste no gráfico de previsão. Para fazer isso, vamos plotar os dados de treino e teste em cores diferentes. Os dados de treino estão representados pelos pontos em cinza e os dados de teste são aqueles pontos em preto após a linha cinza na vertical.

```
fig = modelo.plot(previsao, figsize=(18,9));
ax = fig.add_subplot()
plt.title('Previsão das Temperaturas Médias da Superfície do Solo no
Brasil \nDados de Treino e Teste',
loc = 'left', fontsize = 35)
plt.plot(df_treino['ds'],df_treino['y'],'.',color='grey')
plt.plot(df_teste['ds'],df_teste['y'],'.',color='black')
ax.axvline(x=df_treino['ds'].max(), c='dimgray', lw=3, alpha=0.9, ls='-')
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

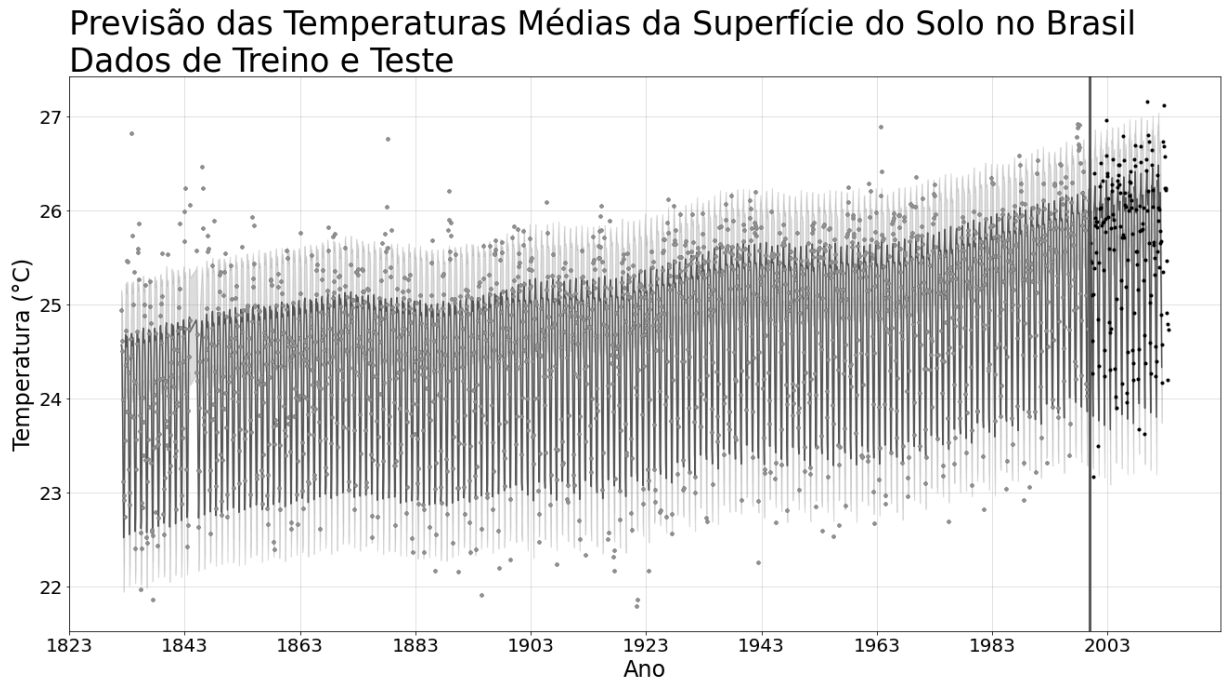

Figura 3.1: Gráfico de previsão contendo dados de teste e treino.

Agora temos os dados de treino e teste no mesmo gráfico. Analisando esse gráfico é possível observar que os dados de teste estão, no geral, bem aderidos à nossa previsão, pois temos poucos pontos para fora. Vamos checar as componentes:

```
modelo.plot_components(previsao, figsize=(10,6));
```

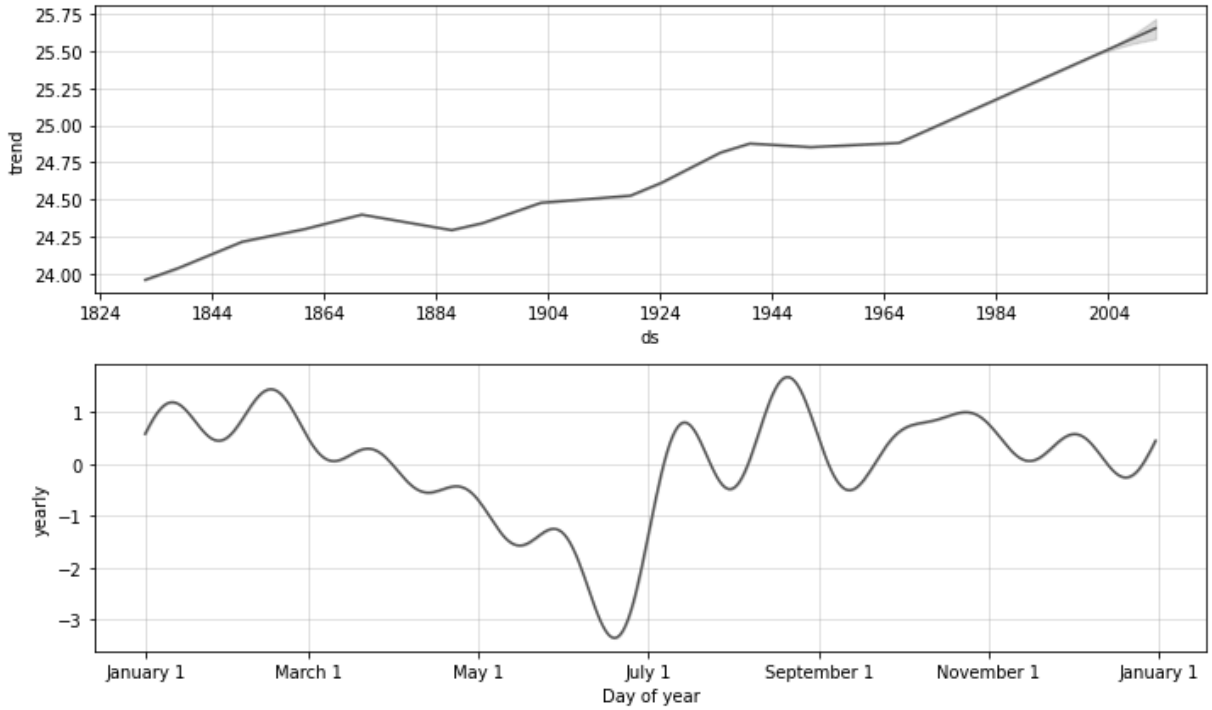

Figura 3.2: Gráficos das componentes.

Novamente, no primeiro gráfico (*trend*), vemos aquela tendência de aumento das temperaturas. Em cinza claro, no final da tendência, temos o nosso intervalo de confiança.

3.1 Previsão de crescimento

Por padrão, o Prophet usa um modelo linear para sua previsão. Ao prever o crescimento, geralmente há algum ponto máximo alcançável: tamanho total do mercado, tamanho total da população etc. Isso é chamado de capacidade de carga, e a previsão deve saturar neste ponto.

Além do crescimento linear, o Prophet permite que você faça previsões usando um modelo de tendência de crescimento logístico, com uma capacidade de carga especificada.

Para entender sobre o crescimento logístico, vamos primeiro entender o crescimento exponencial. Quando falamos em crescimento exponencial, pensamos em algo que depende de quantidades infinitas de recursos (que tendem a não existir no mundo real). O crescimento exponencial pode acontecer por um tempo, se houver poucos indivíduos e muitos recursos. Porém, quando o número de indivíduos aumenta o suficiente, os recursos começam a se esgotar, diminuindo a taxa de crescimento.

Eventualmente, a taxa de crescimento se estabilizará, fazendo uma curva em forma de S. O tamanho da população em que se estabiliza, que representa o tamanho máximo da população que um determinado ambiente pode suportar, é chamado de *capacidade de carga*. Essa curva em forma de S é o crescimento logístico.

Para trazer um exemplo bem próximo da nossa realidade, vamos utilizar dados da pandemia da COVID-19. Os dados foram obtidos da página **Brasil.io** (https://brasil.io/home/), e apresentam o número de casos acumulados de 25 de fevereiro de 2020 até 31 de janeiro de 2022, na Cidade de São Paulo. Esses dados estão disponíveis também no repositório do GitHub deste livro no link: https://github.com/allanspadini/livro-val-allan/blob/main/covid_sp.csv.

O crescimento logístico será utilizado para estimar diretamente um modelo para casos cumulativos. Então, vamos ler o nosso conjunto de dados:

```
covid =
pd.read_csv('https://github.com/vqrca/covidSP/blob/main/covid_sp.csv?
raw=True')
covid.head(3)
```

Na tabela a seguir temos as três primeiras linhas e as principais colunas do dataset:

index	date	state	city	last_available_confirmed
0	2022-01-31	SP	São Paulo	976214
1	2022-01-30	SP	São Paulo	976214
2	2022-01-29	SP	São Paulo	976214

Nossas informações de interesse aqui são as colunas `date`, que apresenta as datas, e a coluna `last_available_confirmed`, que traz o número de casos acumulados. Como já foi explicado no capítulo 1, nós precisamos indicar no DataFrame que estamos lidando com datas e, para isso, usamos `.to_datetime` da Pandas. Aqui nós também adicionamos o parâmetro `format`:

```
covid['date'] = pd.to_datetime(covid['date'], format='%Y-%m-%d')
```

Antes de realizar uma previsão, vamos entender como esses dados se comportam através de um gráfico:

```
fig, ax = plt.subplots(figsize = (14, 6))
sns.lineplot(x='date', y='last_available_confirmed', data=covid, linewidth
= 4, color='grey')
plt.title('SARS-CoV-2: Casos Acumulados na Cidade de São Paulo', fontsize
= 25)
plt.ylabel('Número de Casos', fontsize= 20)
plt.xlabel('Período (Ano/Mês)', fontsize= 20)
plt.ticklabel_format(style='plain', axis='y') #style='plain'desativa a
notação científica
plt.show()
```

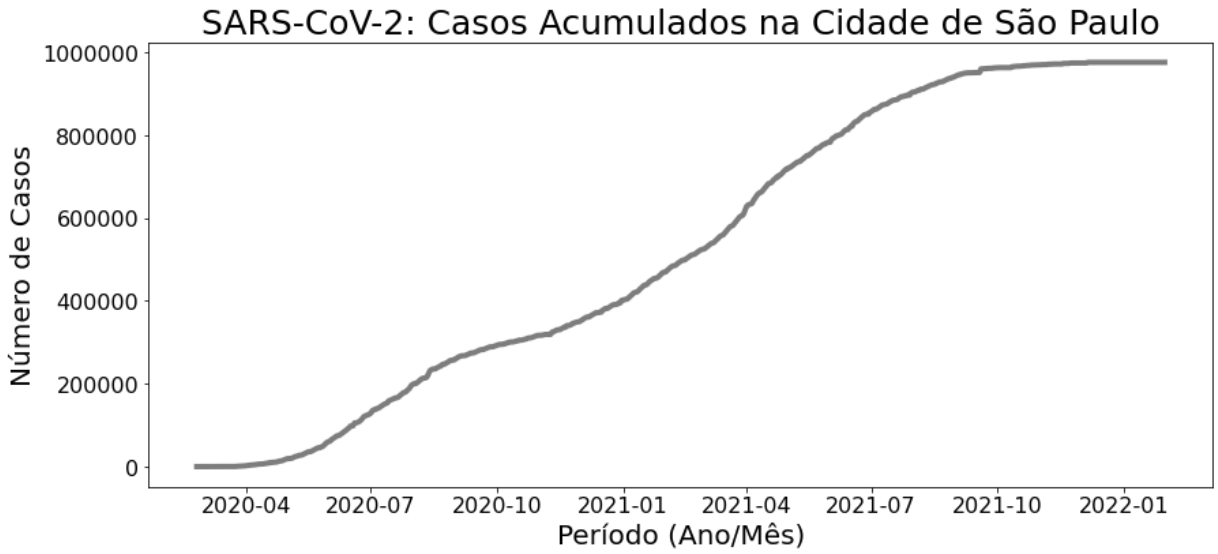

Figura 3.3: Gráfico de casos acumulados de SARS-CoV-2 na cidade de São Paulo.

Nesse gráfico, no eixo `x` temos os períodos de fevereiro de 2020 até janeiro de 2022 e no eixo `y`, o número de casos acumulados. É possível observar que a curva está apresentando um formato parecido ao de S. Nós tivemos fases de crescimento exponencial ao longo da pandemia. Porém, após uma série de medidas e início da vacinação da população, é notável que há uma tendência de estabilização. Vamos seguir com a previsão?

Antes de iniciar a previsão, é necessário criar um novo DataFrame, que será chamado de `df_covid` e nele vamos incluir os dados de `ds` e `y`:

```
df_covid = pd.DataFrame()
df_covid['ds'] = covid['date']
df_covid['y'] = covid['last_available_confirmed']
```

Devemos especificar a capacidade de carga, criando uma coluna chamada `cap`. Aqui assumimos um valor específico, que neste caso será igual a 1000000, pois os valores mais altos dos nossos dados estão próximos a este valor:

```
df_covid['cap'] = 1000000
df_covid.head()
```

Aqui estamos visualizando as primeiras 5 linhas do DataFrame, que agora contém a coluna `cap`.

index	ds	y	cap
0	2022-01-31	976214	1000000
1	2022-01-30	976214	1000000
2	2022-01-29	976214	1000000
3	2022-01-28	976214	1000000
4	2022-01-27	976214	1000000

Por fim, vamos rodar o modelo utilizando o crescimento logístico. Para isso, passamos um argumento adicional: `growth='logistic'`. Na previsão, mantemos a capacidade constante no mesmo valor do histórico e estamos prevendo 30 dias:

```
modelo = Prophet(growth='logistic')
modelo.fit(df_covid)
dataFramefuturo = modelo.make_future_dataframe(periods=30)
dataFramefuturo['cap'] = 1000000
previsao = modelo.predict(dataFramefuturo)
```

Em seguida, plotamos o gráfico da previsão:

```
modelo.plot(previsao, figsize=(18,9));
plt.title('SARS-CoV-2: Previsão de Casos Acumulados na Cidade de São
Paulo', loc = 'left', fontsize = 35)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Mês/Ano', fontsize=24)
plt.ylabel('Número de Casos', fontsize=24)
plt.ticklabel_format(style='plain', axis='y')
plt.show()
```

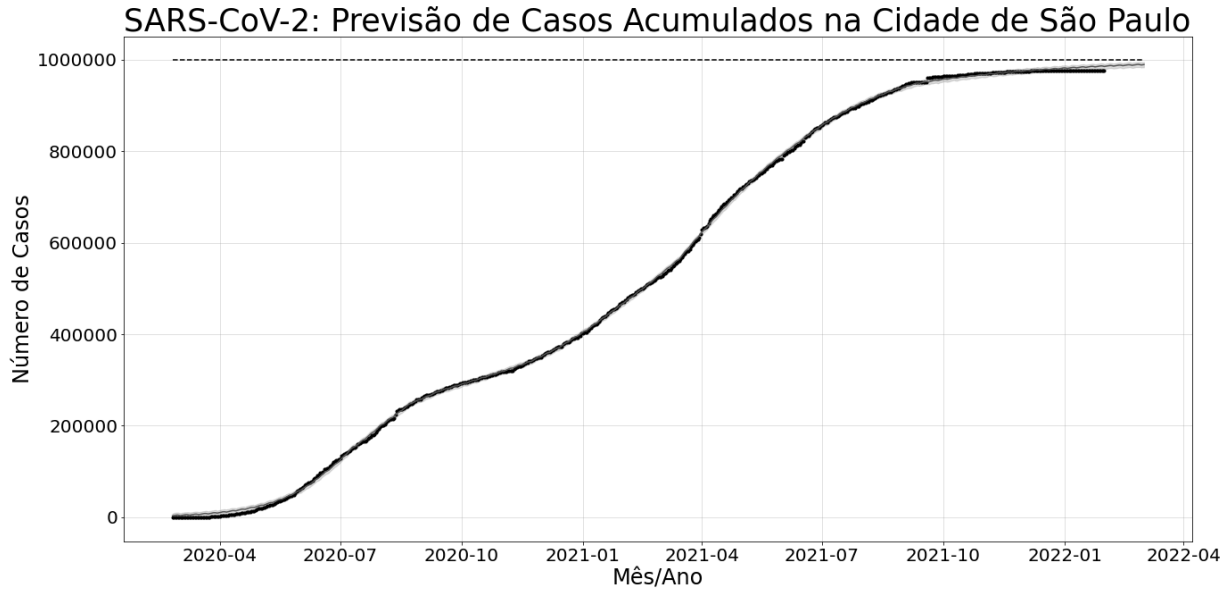

Figura 3.4: Gráfico de previsão do crescimento logístico de casos acumulados de SARS-CoV-2 na cidade de São Paulo.

Nesse gráfico, a linha tracejada na parte superior indica a saturação. Após a curva em preto, que contém os nossos dados, temos a região prevista em cinza, que está bem próxima à capacidade de carga delimitada aqui.

3.2 Changepoints

O Prophet também permite adicionar *Changepoints* ao modelo. *Changepoints* são pontos de mudança, que permitem identificar em quais datas ocorreram possíveis mudanças no comportamento da nossa série temporal. Por padrão, o Prophet detecta automaticamente pontos de mudança e permitirá que a tendência se adapte de forma adequada. No entanto, se você deseja ter um controle mais preciso sobre este processo, existem vários argumentos de entrada que você pode usar.

Para conseguir plotar os gráficos contendo esses pontos de mudança, é necessário realizar a seguinte importação:

```
from fbprophet.plot import add_changepoints_to_plot
```

Vamos retornar ao projeto de mudanças climáticas onde nosso primeiro gráfico será a adição dos `Changepoints` de forma automática. Então, basta executar o seguinte comando:

```
fig = modelo.plot(previsao, figsize=(18,9));
a = add_changepoints_to_plot(fig.gca(), modelo, previsao)
plt.xlabel('Ano')
plt.ylabel('Temperatura (°C)’)
plt.show()
```

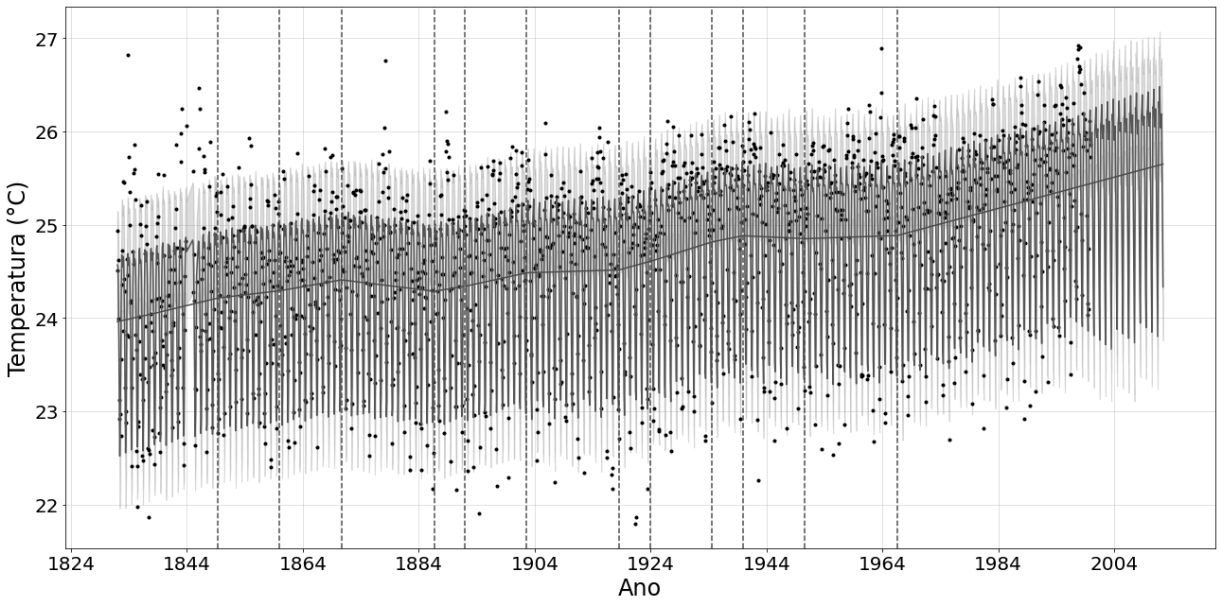

Figura 3.5: Gráfico de previsão contendo Changepoints.

Note que o Prophet especifica alguns pontos de mudança potenciais que são colocados uniformemente nos primeiros 80% da série temporal. As linhas verticais nessa figura indicam onde os pontos de mudança potenciais foram colocados.

Analisando as componentes, não temos nenhuma mudança na tendência:

```
modelo.plot_components(previsao, figsize=(10,6));
```

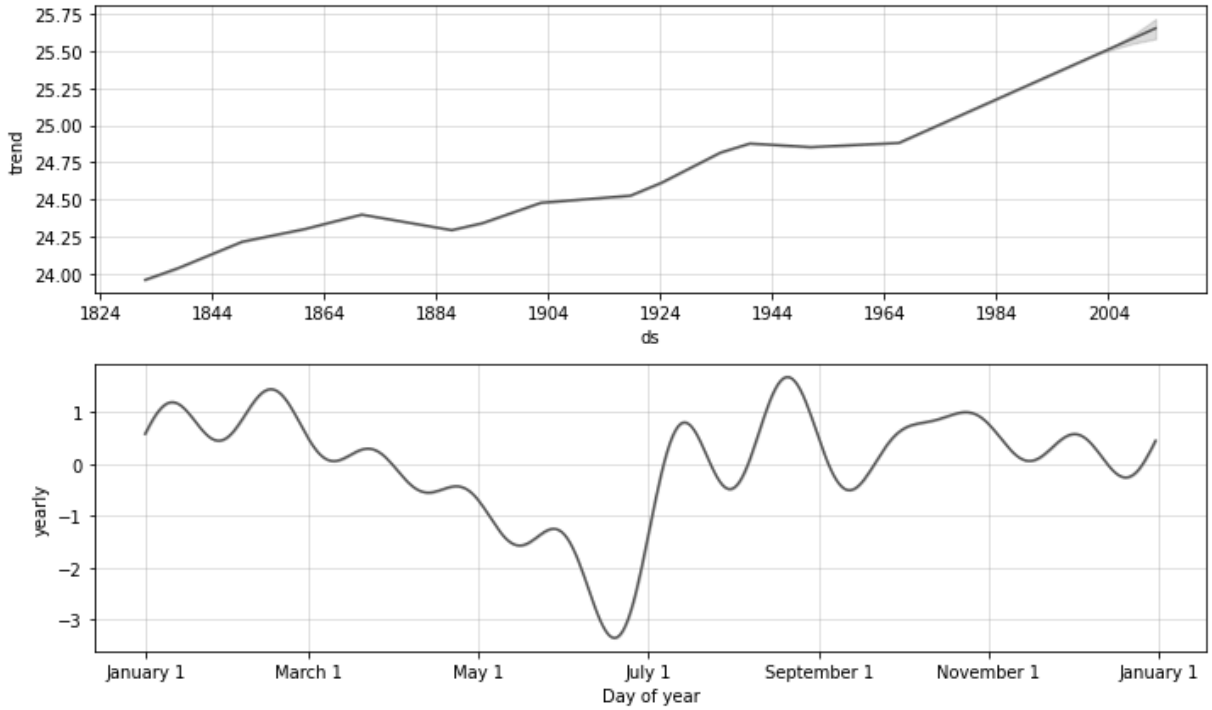

Figura 3.6: Gráficos das componentes.

Alterando o número de Changepoints

Porém, nós podemos alterar a quantidade de `Changepoints` . Para alterar a quantidade de `changepoints` o argumento `n_changepoints` é adicionado ao modelo. Por exemplo, vamos adicionar apenas 5 `changepoints` e analisar o que vai acontecer.

```
modelo = Prophet(n_changepoints=5)
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=15, freq = 'M')
previsao = modelo.predict(dataFramefuturo)
```

Gráfico da previsão:

```
fig = modelo.plot(previsao, figsize=(18,9));
a = add_changepoints_to_plot(fig.gca(), modelo, previsao)
plt.xlabel('Ano')
plt.ylabel('Temperatura (°C)’)
plt.show()
```

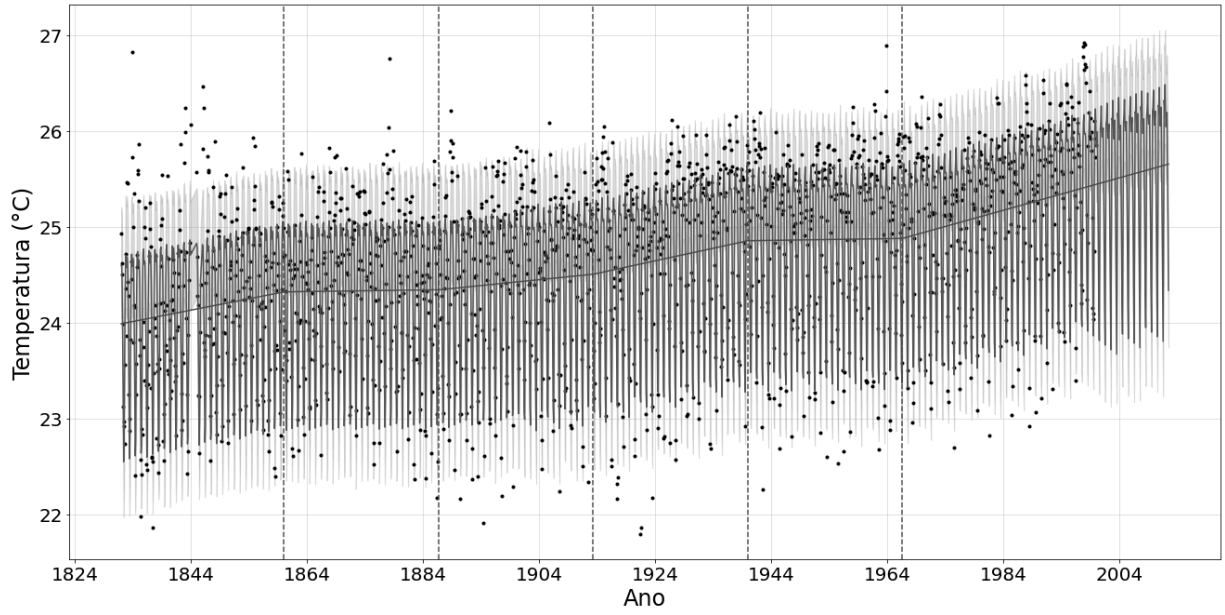

Figura 3.7: Gráfico de previsão contendo Changepoints.

Temos apenas 5 pontos de mudança, mas, para evidenciar melhor o que mudou, vamos analisar o gráfico das componentes:

```
modelo.plot_components(previsao, figsize=(10,6));
```

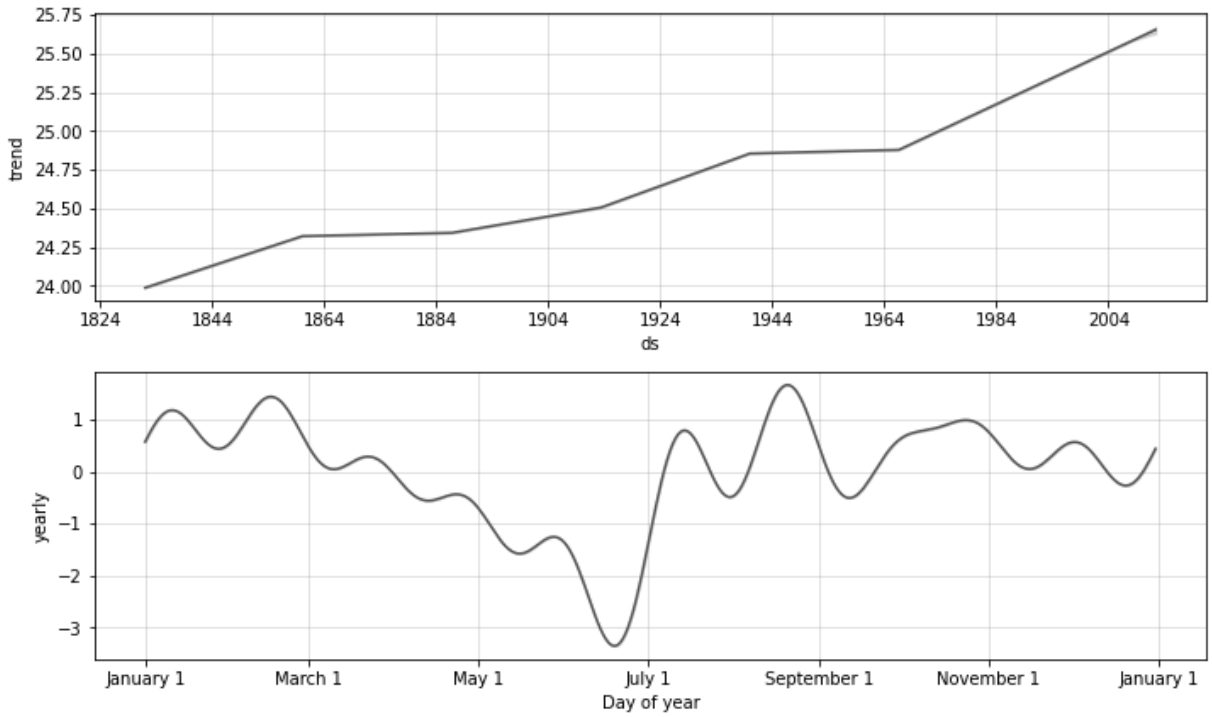

Figura 3.8: Gráficos das componentes.

Como consequência, nossa linha de tendência agora possui menos pontos de mudança. Não ter tantos *Changepoints* é algo que diminui a possibilidade de *overfitting*.

> *Overfitting* é um problema muito comum quando trabalhamos com *Machine Learning* ou na previsão de séries temporais. Esse termo significa um sobreajuste, ou seja, é o risco de que seu modelo de previsão seja preciso apenas para prever o passado, e não para prever bem o futuro. Um bom modelo de previsão deve ser bom para prever os dados que você não possui!

changepoint_range

Além de adicionar os `changepoints` nós também podemos alterar onde eles são adicionados. Para fazer isso, podemos usar o argumento `changepoint_range` . O padrão desse argumento é 0.80. Isso significa que o modelo não se ajustará a nenhuma mudança de

tendência nos últimos 20% da série temporal para manter a tendência geral da série. Isso é algo conservador, para evitar o *overfitting* do modelo. Vamos alterar esse parâmetro para 0.90 para observar a mudança:

```
modelo = Prophet(n_changepoints=5, changepoint_range=0.90)
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=15, freq = 'M')
previsao = modelo.predict(dataFramefuturo)
```

Gráfico da previsão:

```
fig = modelo.plot(previsao, figsize=(18,9));
a = add_changepoints_to_plot(fig.gca(), modelo, previsao)
plt.xlabel('Ano')
plt.ylabel('Temperatura (°C)’)
plt.show()
```

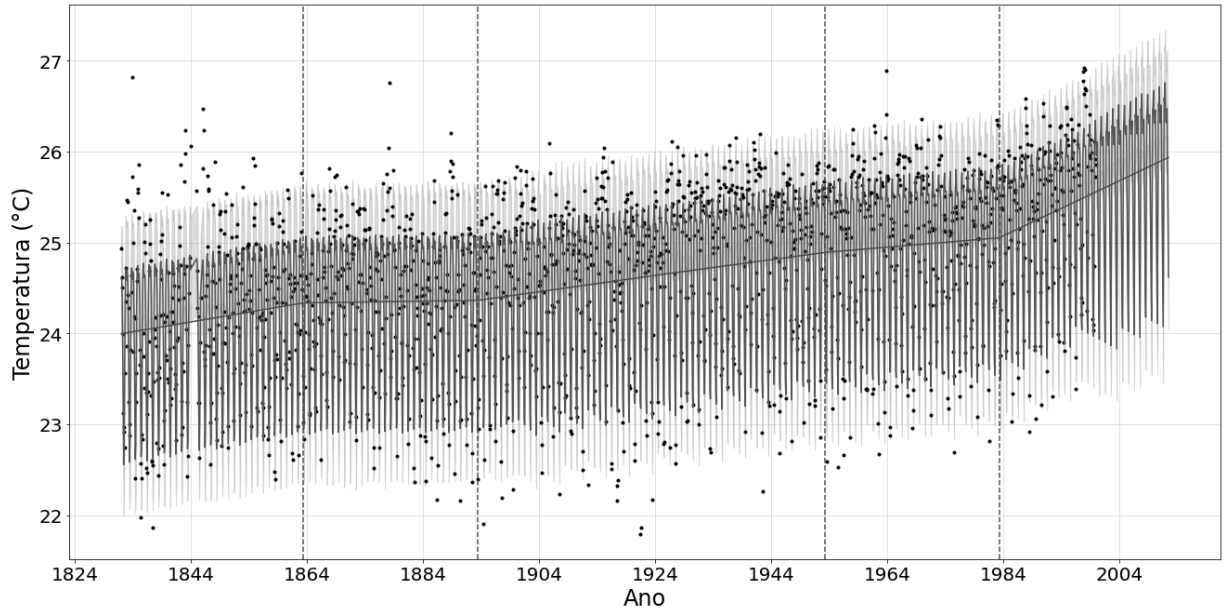

Figura 3.9: Gráfico de previsão contendo Changepoints.

Nesse caso, temos `changepoints` adicionados perto do final da nossa série temporal. Vamos plotar as componentes para analisar melhor a tendência:

```
modelo.plot_components(previsao, figsize=(10,6));
```

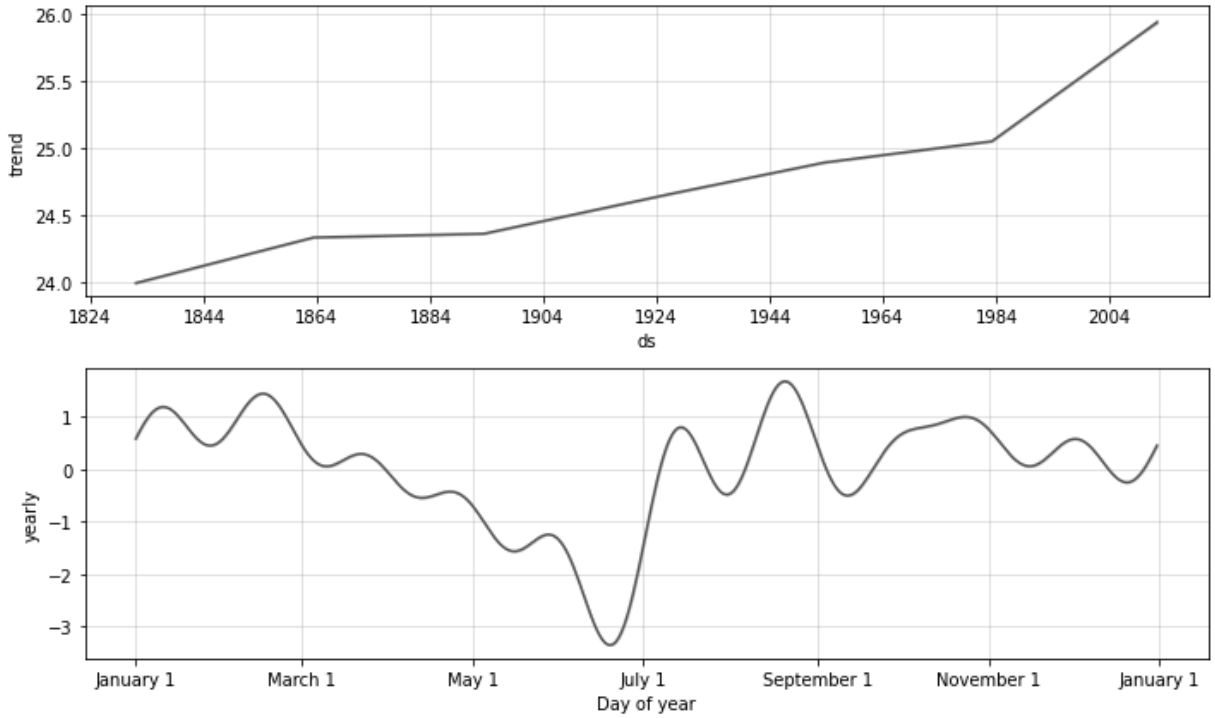

Figura 3.10: Gráficos das componentes.

Nossa tendência está um pouco diferente agora! Temos poucas mudanças e a mais brusca agora está em 1984. Se nós voltarmos ao gráfico 3.8, antes de alterar este parâmetro, vamos notar que anteriormente esse `changepoint` estava próximo do ano 1964.

changepoint_prior_scale

Outra coisa que podemos fazer é ajustar a flexibilidade da tendência. Se as mudanças de tendência estão sendo superajustadas (flexibilidade demais) ou subajustadas (flexibilidade insuficiente), você pode ajustar o argumento de entrada `changepoint_prior_scale`. Por padrão, esse parâmetro é definido como 0.05. Aumentá-lo tornará a tendência mais flexível.

Vamos aumentar a flexibilidade para 10 para que seja possível notar a diferença. Os outros parâmetros serão mantidos como *default*:

```
modelo = Prophet(changepoint_prior_scale=10)
modelo.fit(df)
```

```
dataFramefuturo = modelo.make_future_dataframe(periods=150, freq = 'M')
previsao = modelo.predict(dataFramefuturo)
```

Gráfico da previsão:

```
fig = modelo.plot(previsao, figsize=(18,9));
a = add_changepoints_to_plot(fig.gca(), modelo, previsao)
plt.xlabel('Ano')
plt.ylabel('Temperatura (°C)’)
plt.show()
```

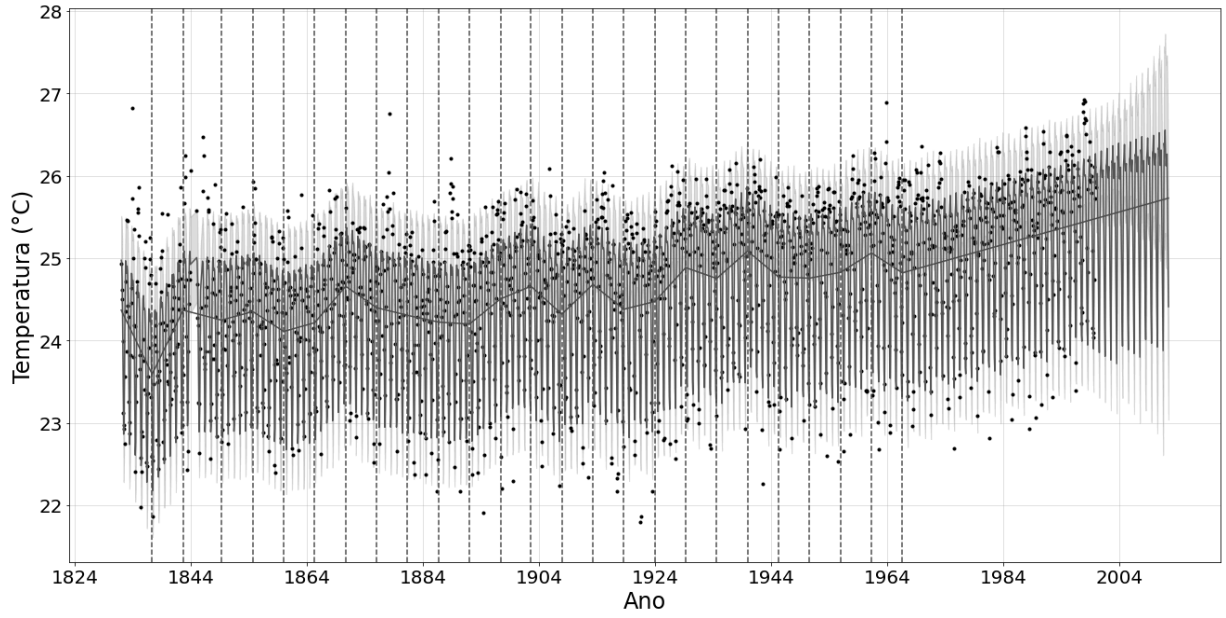

Figura 3.11: Gráfico de previsão contendo *changepoints*, com changepoint_prior_scale=10.

Está bem evidente que o modelo está mais flexível. Se nós analisarmos a tendência nas componentes isso vai ficar ainda mais claro:

```
modelo.plot_components(previsao, figsize=(10,6));
```

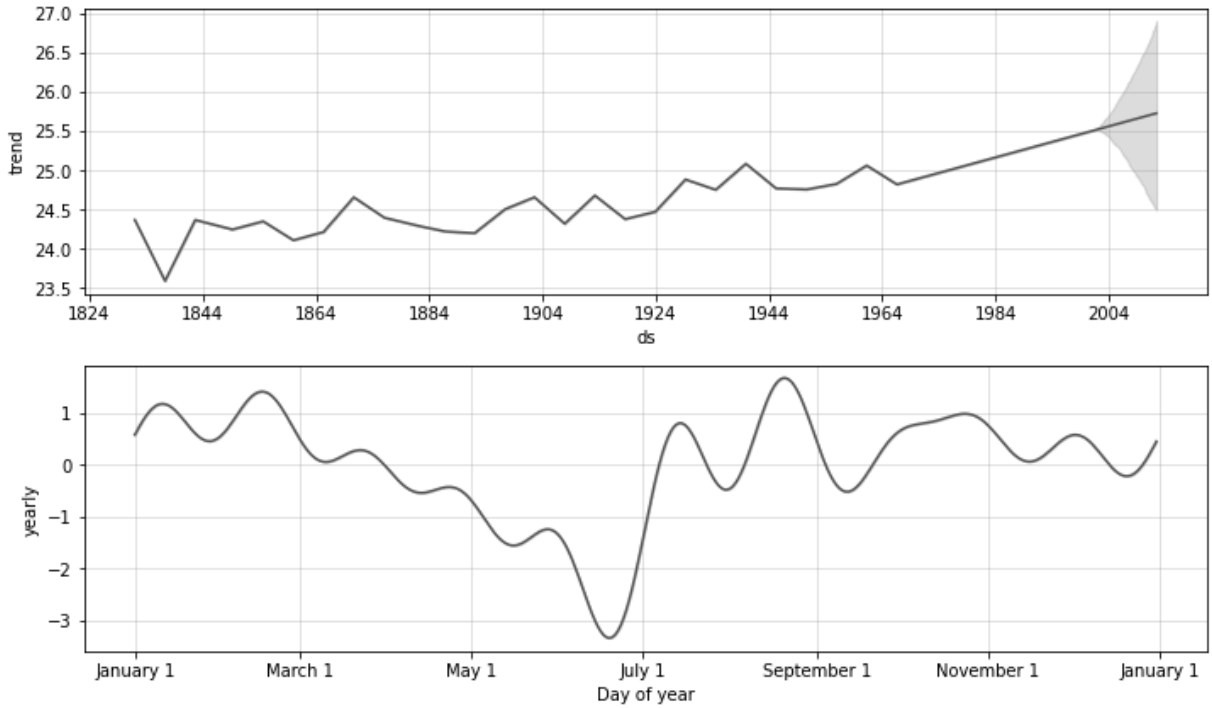

Figura 3.12: Gráficos das componentes.

A flexibilidade está bem alta e temos diversos momentos em que a tendência está tentando se adaptar aos pontos de mudança. Uma coisa importante quando alteramos parâmetros como esse é que o intervalo de confiança em torno dos valores previstos fica muito elevado. Observe a faixa em cinza no final da tendência: ela está muito mais ampla comparada aos exemplos anteriores. É importante destacar, também, que quando aumentamos demais a flexibilidade podemos gerar *overfitting*.

Datas específicas

Também podemos especificar a data dos `changepoints`. Se você souber uma data em que algo influenciou e gerou uma mudança, esta data pode ser adicionada ao modelo, no argumento `changepoints`:

```
modelo = Prophet(changepoints=['1980-01-01'])
modelo.fit(df)
```

```
dataFramefuturo = modelo.make_future_dataframe(periods=150, freq = 'M')
previsao = modelo.predict(dataFramefuturo)
```

Gráfico da previsão:

```
fig = modelo.plot(previsao, figsize=(18,9));
a = add_changepoints_to_plot(fig.gca(), modelo, previsao)
plt.xlabel('Ano')
plt.ylabel('Temperatura (°C)’)
plt.show()
```

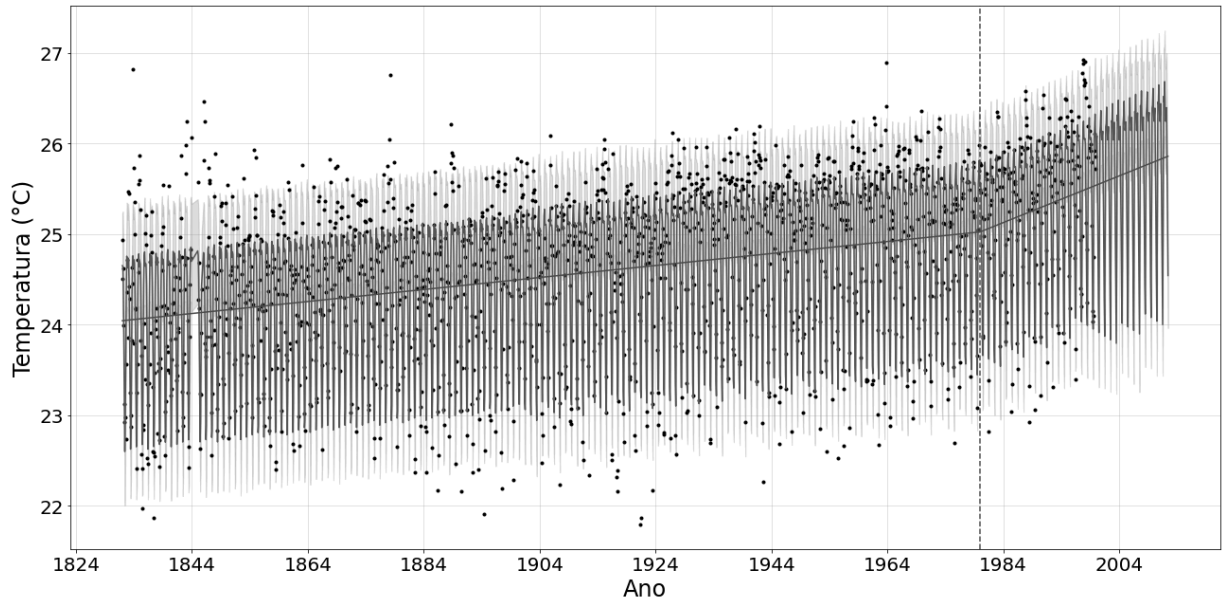

Figura 3.13: Gráfico de previsão contendo Changepoints.

Logo, temos apenas um `changepoint` . Vamos analisar a tendência no gráfico das componentes:

```
modelo.plot_components(previsao, figsize=(10,6));
```

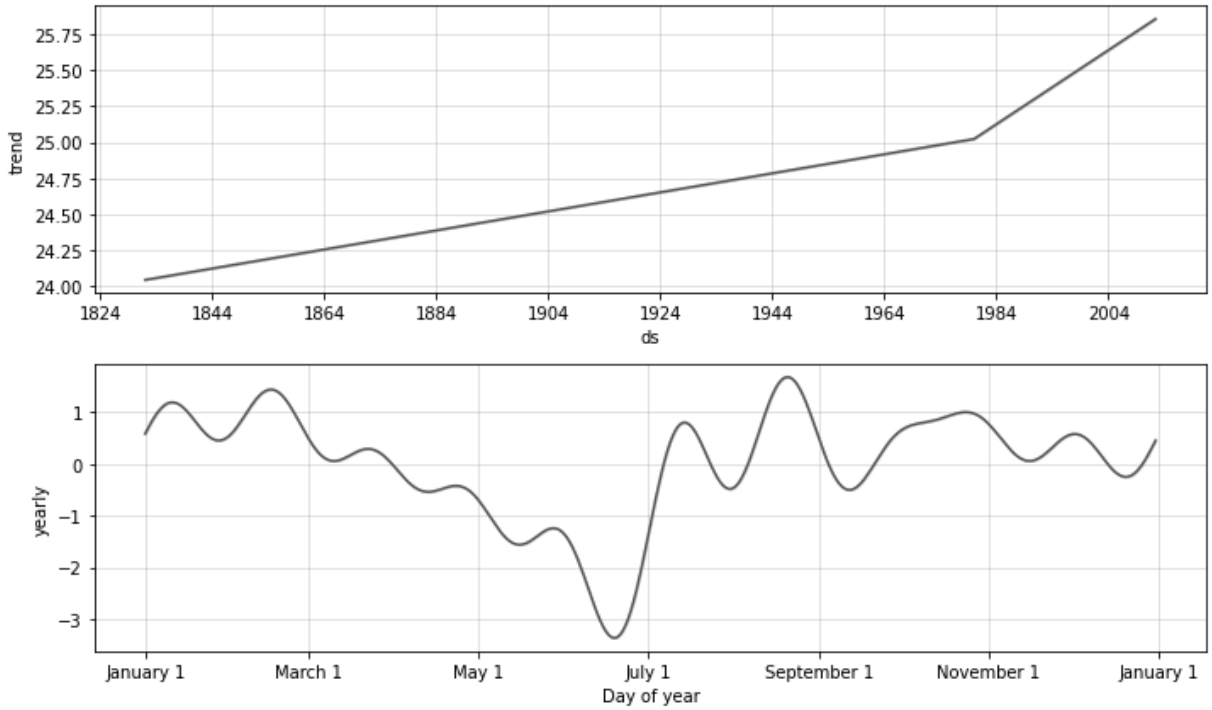

Figura 3.14: Gráficos das componentes.

Se você souber de várias datas, que podem ter influenciado a série temporal, uma lista contendo essas datas pode ser criada e adicionada no argumento `changepoints`. Vamos supor que você esteja trabalhando com um *dataset* de faturamento de uma loja. Nós sabemos, que em alguns feriados ou datas específicas as vendas podem aumentar, certo?! Dia das mães, dia dos pais, Black Friday, natal etc… Em casos como esses, você também pode criar uma lista com essas datas e adicioná-la ao modelo.

Existe algo parecido com as mudanças de tendência, mas que ocorre sempre em determinadas datas. Por exemplo, a temperatura diminui no inverno de todos os anos e alguns tipos de mercados também vendem mais em épocas específicas. Esses fenômenos que se repetem periodicamente são chamados de sazonalidade. Existem tipos específicos e formas de inserir essas informações no Prophet. É sobre esse tópico que vamos trabalhar no capítulo a seguir.

Capítulo 4
Sazonalidade

Com o Prophet, nós também podemos verificar o efeito da sazonalidade. Mas, antes de tudo, o que é sazonalidade? Algo que caracteriza as séries temporais é o fato de que os dados têm um certo ritmo. Esse ritmo pode ser anual, semanal, diário ou até mesmo horário. Quando pensamos no ritmo anual, podemos associá-lo à rotação da Terra em torno do Sol; no ritmo diário, à rotação da Terra em torno de seu próprio eixo. Se você for analisar uma série temporal de dados sobre congestionamento do tráfego, por exemplo, você vai notar que ela vai seguir o ciclo das atividades das pessoas ao longo do dia. Inclusive, o nosso próprio corpo segue ciclos: batimento cardíaco, frequência respiratória e o ritmo circadiano.

O Prophet chama esses ciclos de *sazonalidades*, que podem ser aditivas ou multiplicativas. A sazonalidade aditiva é quando a série apresenta flutuações sazonais mais ou menos constantes, não importando o nível global da série. Enquanto isso, a sazonalidade multiplicativa ocorre quando o tamanho das flutuações sazonais varia dependendo do nível global da série.

Vamos começar analisando a sazonalidade aditiva no nosso modelo. É importante destacar que o Prophet utiliza a sazonalidade aditiva como parâmetro *default*. Mas o argumento `seasonality_mode='additive'` será adicionado mesmo assim, de forma didática:

```
modelo = Prophet(seasonality_mode='additive')
modelo.fit(df)
dataFramefuturo= modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataFramefuturo)
```

Plotando o gráfico:

```
modelo.plot(previsao, figsize=(18,6));
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

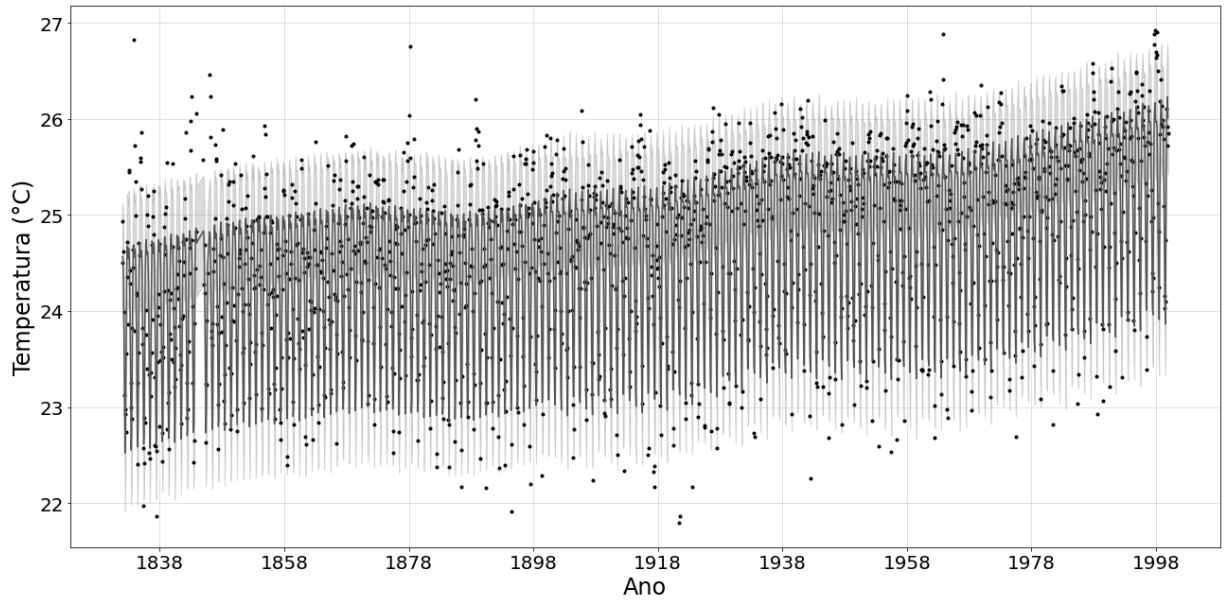

Figura 4.1: Gráfico da previsão com a sazonalidade definida como aditiva.

Esse gráfico mostra o efeito da sazonalidade aditiva e nós podemos utilizar métricas para avaliar como o modelo está performando. No **Sklearn** temos diversas métricas para avaliar modelos de regressão (https://scikit-learn.org/stable/modules/model_evaluation.html).

Vamos importar uma dessas métricas: a `mean_squared_error`. Essa métrica é o erro quadrático médio, ou seja, é a média da diferença entre o valor do `y` real e do `y` previsto ao quadrado. Então, quanto mais próximo de zero o resultado, melhor está o nosso modelo!

Primeiramente, vamos importar a métrica:

```
from sklearn.metrics import mean_squared_error
```

Agora vamos aplicar essa métrica na nossa previsão, nos dados do treino:

```
mean_squared_error(y_true = df['y'],
y_pred=previsao['yhat'][:2000])
```

Nesse caso, o `mean_squared_error` é igual a 0.19. Agora vamos analisar a sazonalidade multiplicativa para comparar os resultados. Para isso, é necessário usar o argumento `seasonality_mode =multiplicative` :

```
modelo = Prophet(seasonality_mode='multiplicative')
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataFramefuturo)
```

Plotando o gráfico:

```
modelo.plot(previsao, figsize=(18,6));
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

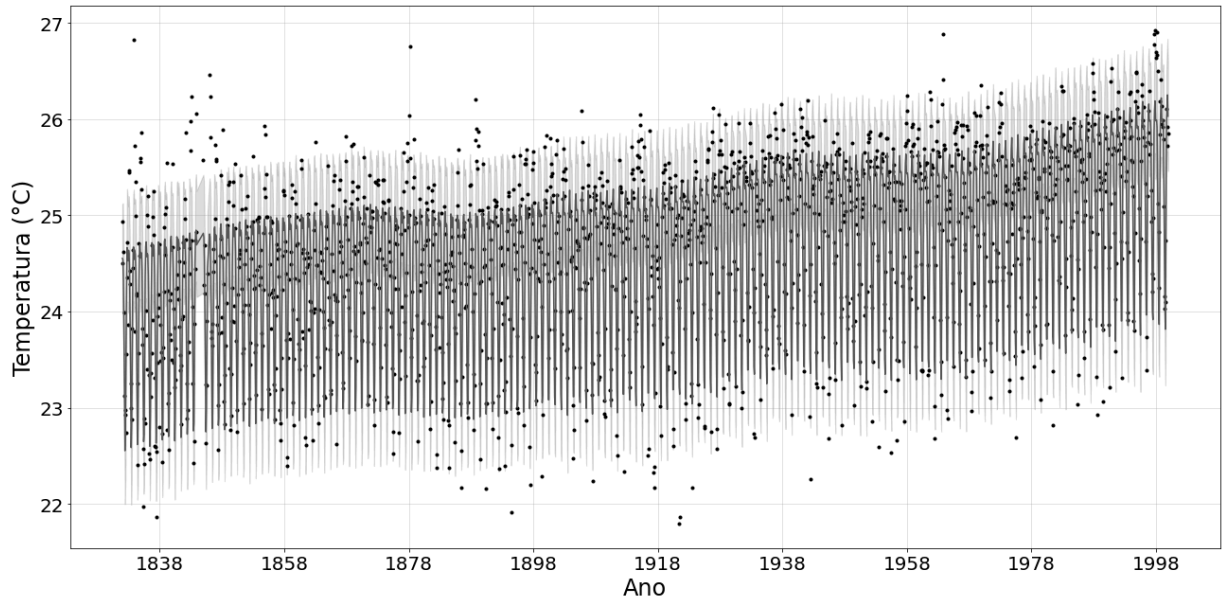

Figura 4.2: Gráfico da previsão com a sazonalidade definida como multiplicativa.

Vamos checar o valor do erro quadrático médio:

```
mean_squared_error(y_true = df['y'],
y_pred=previsao['yhat'][:2000])
```

Bom, o valor também é igual a 0.19 e, para esses dados, especificamente, nós não conseguimos notar a diferença entre as sazonalidades.

4.1 Sazonalidade aditiva x Sazonalidade multiplicativa

Em outros casos pode acontecer de os cenários serem bastante diferentes. Para que você possa entender isso, vamos comparar os dois tipos de sazonalidades utilizando um *dataset* disponibilizado na plataforma **Key2stat** (https://www.key2stats.com/data-set/view/745). Neste *dataset* nós temos as vendas mensais de medicamentos antidiabéticos na Austrália, no período de julho de 1991 a junho de 2008.

Primeiramente, o *dataset* que está disponível no GitHub em: https://github.com/allanspadini/livro-val-allan/blob/main/AusAntidiabeticDrug.csv será carregado.

```
df = pd.read_csv('https://github.com/allanspadini/livro-val-allan/blob/main/AusAntidiabeticDrug.csv?raw=true')
```

Vamos checar as cinco primeiras linhas do DataFrame:

```
df.head()
```

index	ds	y
0	1991-07-01	3.526591
1	1991-08-01	3.180891
2	1991-09-01	3.252221

index	ds	y
3	1991-10-01	3.611003
4	1991-11-01	3.565869

Temos apenas duas colunas: a primeira apresenta as datas e a segunda, a quantidade de vendas. As colunas já estão nomeadas como `ds` e `y`, então já é possível rodar o modelo, sem a necessidade de criar um novo DataFrame. Vamos começar testando a sazonalidade aditiva e adicionar um período de 36 meses de previsão:

```
#aditiva
modelo = Prophet(seasonality_mode='additive')
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=36, freq='M')
previsao = modelo.predict(dataFramefuturo)
```

Plotando o gráfico:

```
modelo.plot(previsao, figsize=(18,9));
plt.title('Sazonalidade Aditiva ', loc = 'left', fontsize = 35)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Vendas', fontsize=24)
plt.show()
```

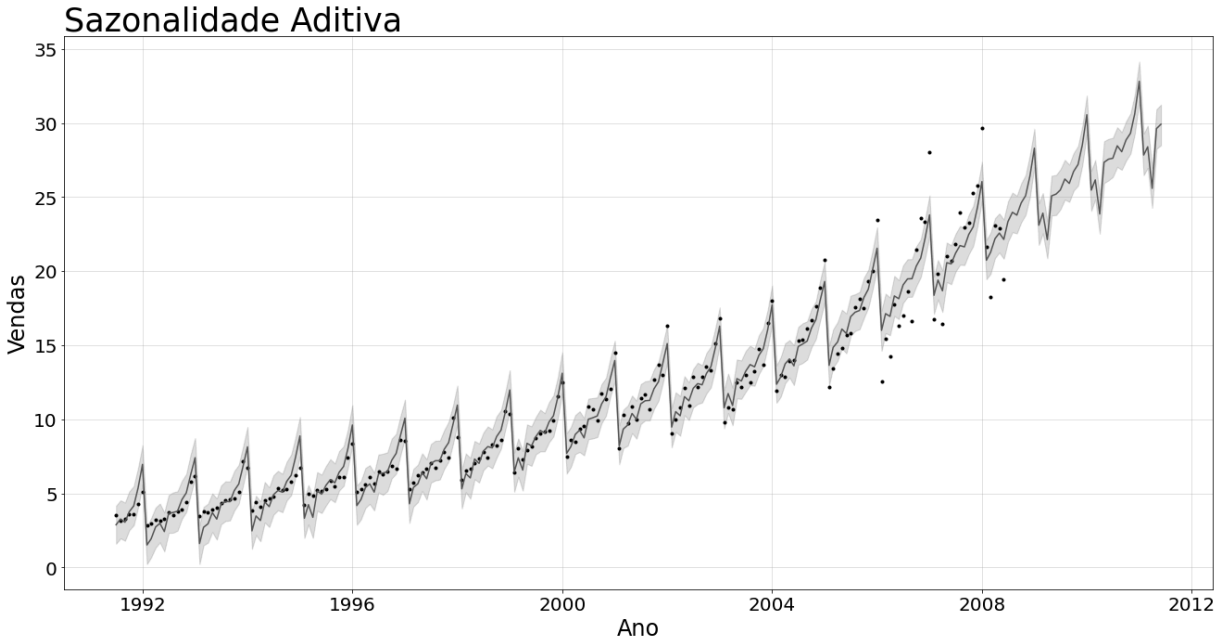

Figura 4.3: Gráfico da previsão com a sazonalidade definida como aditiva.

Como você pode analisar, não há uma constância nos dados, e, por isso, não há um ajuste tão bom da curva, principalmente após 2004, onde temos grandes oscilações sazonais. A propagação sazonal dos dados está aumentando, mas estamos prevendo que seja constante, já que estamos usando a sazonalidade aditiva aqui. Esse é o erro de escolher a sazonalidade aditiva quando a multiplicativa é necessária. Vamos checar o erro quadrático médio nesse caso:

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'][:204])
```

O resultado é um erro quadrático médio igual a 1.05. Vamos executar o modelo novamente, mas desta vez usaremos sazonalidade multiplicativa:

```
#multiplicativa
modelo = Prophet(seasonality_mode='multiplicative')
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=36, freq='M')
previsao = modelo.predict(dataFramefuturo)
```

É possível notar essa mudança refletida no gráfico a seguir:

```
modelo.plot(previsao, figsize=(18,9));
plt.title('Sazonalidade Multiplicativa ', loc = 'left', fontsize = 35)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Vendas', fontsize=24)
plt.show()
```

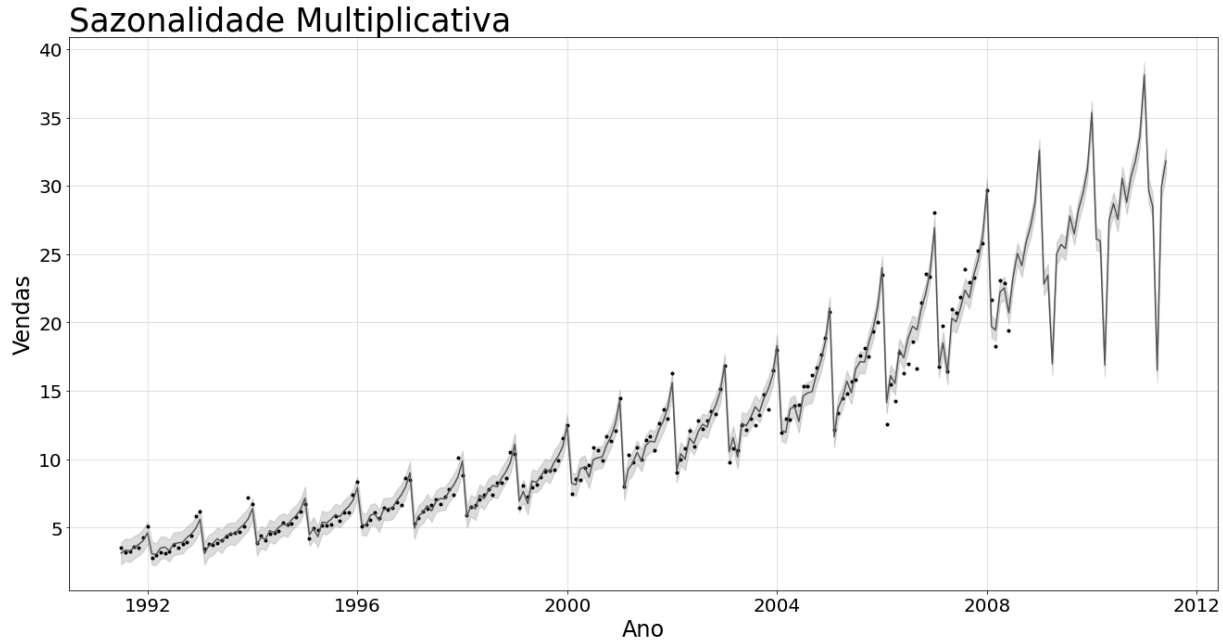

Figura 4.4: Gráfico da previsão com a sazonalidade definida como multiplicativa.

Esse é um ajuste muito melhor! Agora, o Prophet está combinando o crescimento das oscilações de sazonalidade juntamente com o crescimento da tendência geral. É interessante analisar o erro quadrático médio dessa sazonalidade para confirmar que realmente o ajuste está melhor:

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'][:204])
```

Quando executamos esse comando, temos como resultado um erro quadrático médio igual a 0.40. Portanto, o resultado está realmente bem melhor. Além disso, compare as estimativas de erro (a área sombreada em torno da linha sólida) entre as figuras 4.3 e 4.4. Os intervalos de incerteza são mais amplos quando tentamos ajustar a

sazonalidade aditiva a uma série de dados contendo sazonalidade multiplicativa. O Prophet sabe que não tem um bom encaixe no modelo anterior e tem menos certeza sobre suas previsões.

4.2 Ajustando a flexibilidade da sazonalidade

Existe um parâmetro chamado `seasonality_prior_scale`, que controla a flexibilidade da sazonalidade. Um valor alto desse parâmetro permite que a sazonalidade se ajuste a grandes flutuações, já um valor pequeno diminui a magnitude da sazonalidade. O *default* desse parâmetro é 10, que basicamente não aplica nenhuma mudança.

Segundo a documentação do Prophet um intervalo razoável para ajuste provavelmente é entre 0.01 e 10. Quando definimos o parâmetro para 0.01, nós descobrimos que a magnitude da sazonalidade é forçada a ser menor.

Vamos testar esse valor nos nossos dados, adicionando o parâmetro `seasonality_prior_scale=0.01`:

```
modelo = Prophet(seasonality_mode='multiplicative',
seasonality_prior_scale=0.01)
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=36, freq='M')
previsao = modelo.predict(dataFramefuturo)
```

Plotando o gráfico:

```
modelo.plot(previsao, figsize=(18,9));
plt.title('Sazonalidade Multiplicativa ', loc = 'left', fontsize = 35)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Vendas', fontsize=24)
plt.show()
```

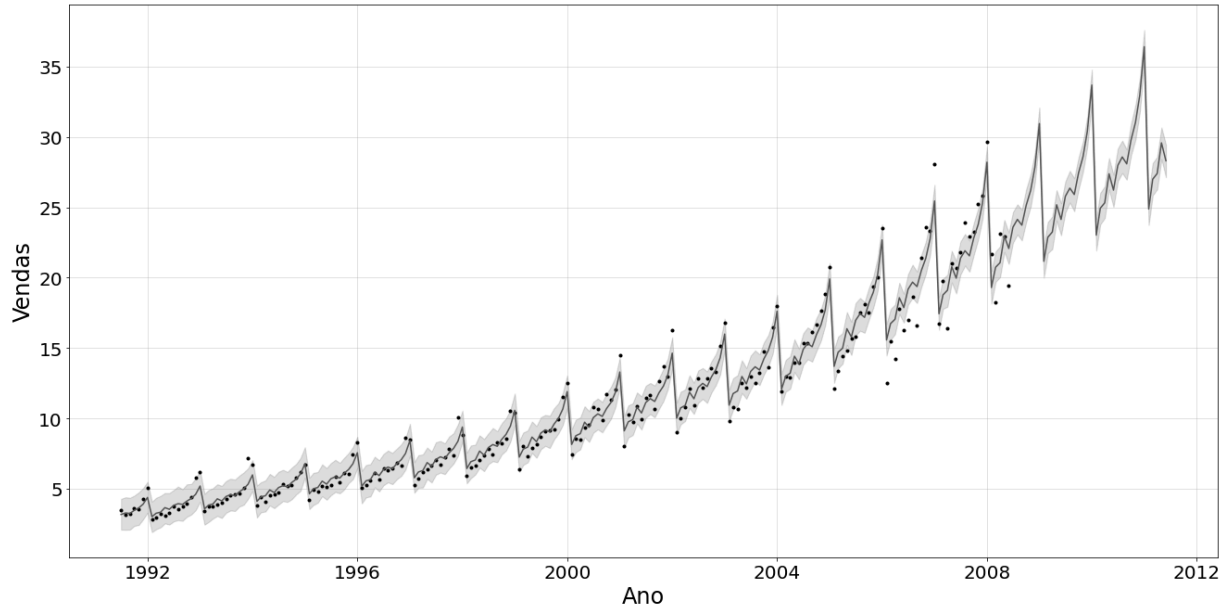

Figura 4.5: Gráfico da previsão com a sazonalidade definida como multiplicativa e adição do parâmetro seasonality_prior_scale=0.01.

É possível observar algumas mudanças entre 2004-2008 comparando esse gráfico e o gráfico 4.4, pois o ajuste dos pontos parecia melhor antes. Podemos analisar novamente a métrica do erro quadrático médio:

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'][:204])
```

Dessa forma, o valor do erro quadrático médio é igual a 0.75. Portanto, o nosso resultado anterior com o valor de `seasonality_prior_scale` *default* é realmente melhor.

4.3 Série de Fourier para sazonalidades

Discutimos sobre os dois tipos de sazonalidade e agora vamos explorar as Séries de Fourier.

As Séries de Fourier foram introduzidas por Jean Baptiste Joseph Fourier no início de 1800 para investigar a Teoria do calor. Uma Série de Fourier é a expansão de uma função periódica f(x) em termos de uma soma infinita de senos e cossenos. As Séries de Fourier fazem uso das relações de ortogonalidade das funções seno e cosseno. A aplicação da Série de Fourier é, portanto, uma análise harmônica extremamente útil para quebrar uma função periódica arbitrária em um conjunto de termos simples que podem ser conectados, resolvidos de forma individual e então recombinados para obter a solução de um problema. Mais detalhes sobre as Séries de Fourier podem ser encontrados no capítulo 3 do livro Análise de Fourier, disponível em: https://www.ufrgs.br/reamat/TransformadasIntegrais/livro-af/livro.pdf.

Por exemplo, aqui temos o somatório de uma série de senos no intervalo `[0,1]` :

$$1 = \sum_{n-odd}^{\infty} \frac{4}{n\pi} \sin(n\pi x)$$

Figura 4.6: Fórmula.

Na imagem a seguir temos dois gráficos que são obtidos como resultado desse cálculo. Na esquerda temos o número de termos N=3 e na direita, o N=11:

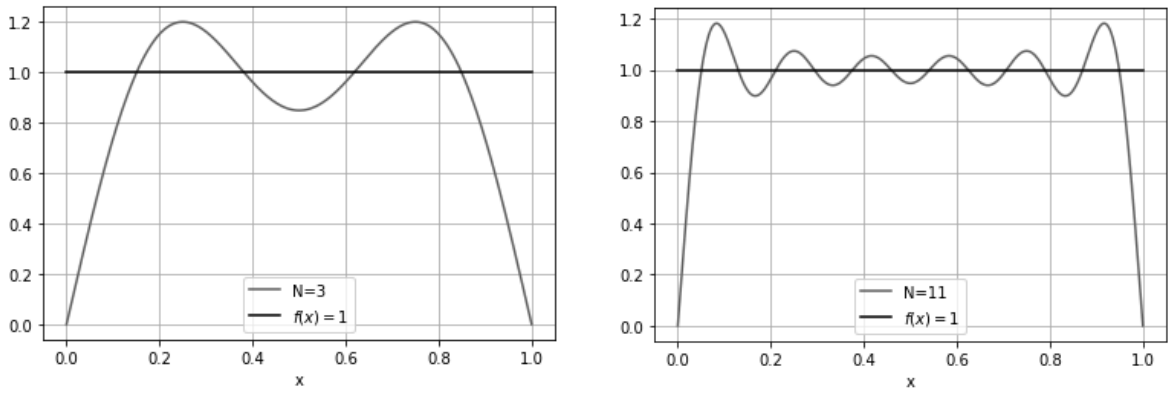

Figura 4.7: Gráficos obtidos após cálculo do Seno de uma Série Fourier no intervalo [0,1] para os termo 3 e 11, respectivamente.

A sazonalidade é estimada usando a soma parcial da Série de Fourier. Então, o número de termos na soma parcial é um parâmetro que determina a rapidez com que a sazonalidade pode mudar. Para ilustrar isso, considere a previsão dos dados anteriores obtidos com a sazonalidade multiplicativa.

Vamos analisar somente o gráfico da sazonalidade anual aqui, então podemos fazer a seguinte importação antes de começar:

```
from fbprophet.plot import plot_yearly
```

A Série de Fourier *default* para sazonalidade anual é 10, o que produz este ajuste:

```
modelo = Prophet(seasonality_mode='multiplicative').fit(df)
fig = plot_yearly(modelo,figsize=(12,8));
```

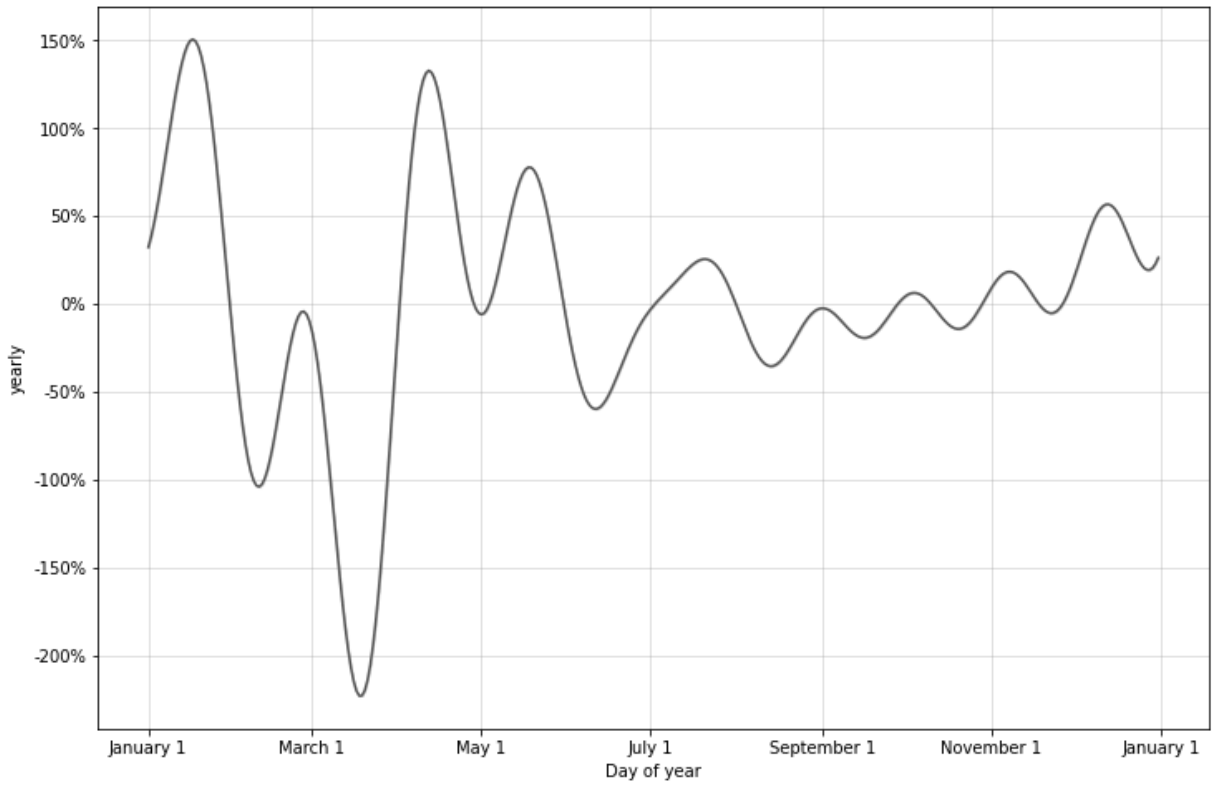

Figura 4.8: Sazonalidade anual com Série de Fourier default.

Os valores *default* geralmente são apropriados, mas podem ser aumentados quando a sazonalidade precisa se ajustar a alterações de frequência mais alta e geralmente são menos suaves. A Série de Fourier pode ser especificada para cada sazonalidade incorporada ao instanciar o modelo por meio do argumento `yearly_seasonality`, que aqui está sendo aumentada para 20:

```
modelo = Prophet(yearly_seasonality=20,
seasonality_mode='multiplicative').fit(df)
fig = plot_yearly(modelo,figsize=(12,8));
```

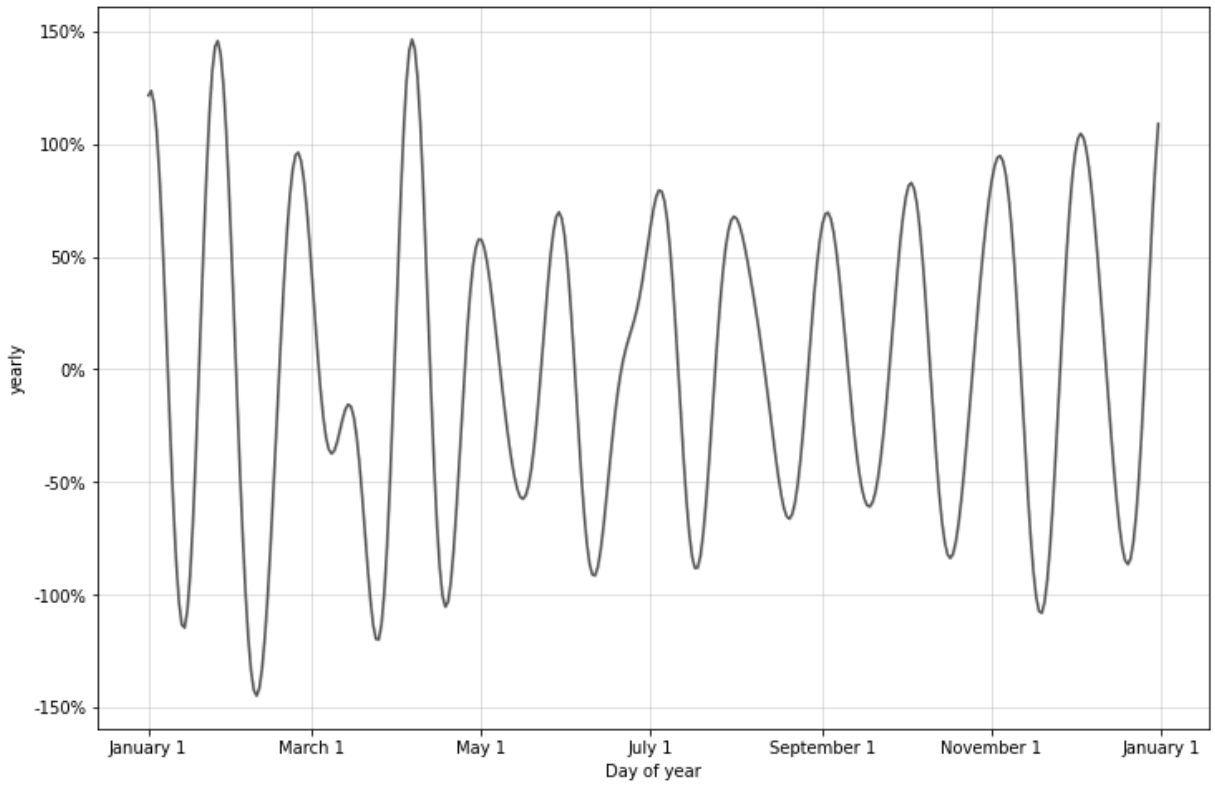

Figura 4.9: Sazonalidade anual com Série de Fourier ajustada para 20.

Portanto, aumentar o número de termos de Fourier permite que a sazonalidade se ajuste a ciclos de mudança mais rápidos. Porém, é importante ressaltar que fazer isso também pode levar a *overfitting*, pois N termos de Fourier correspondem a 2N variáveis usadas para modelar o ciclo.

4.4 Especificando sazonalidades personalizadas

Por padrão, o Prophet ajustará as sazonalidades semanais e anuais, se a série temporal tiver mais de dois ciclos. Ele também ajustará a sazonalidade diária para uma série temporal subdiária. É possível adicionar manualmente outras sazonalidades (mensal, trimestral, horária) usando o método `add_seasonality`.

As entradas para essa função são um nome para a sazonalidade, o período da sazonalidade em dias e a Série de Fourier para a sazonalidade. Segundo a documentação do Prophet, a Série de Fourier igual a 3 é adequada para sazonalidade semanal. Já quando trabalhamos com a sazonalidade anual, a Série de Fourier é igual a 10.

Para personalizar as sazonalidades vamos utilizar outro *dataset* que foi obtido no **Kaggle** (https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset). Esse *dataset* contém a contagem horária e diária de bicicletas alugadas entre os anos de 2015 e 2017 no sistema de bicicletas compartilhadas em Londres. Os dados estão disponíveis no GitHub no repositório do livro em https://github.com/allanspadini/livro-val-allan/blob/main/london_merged.csv.

Então, para iniciar vamos carregar os dados:

```
dados_bikes =  pd.read_csv('https://github.com/allanspadini/livro-val-allan/blob/main/london_merged.csv?raw=true')
dados_bikes.head()
```

Analisando as primeiras linhas temos as seguintes informações:

index	timestamp	cnt	...	season
0	2015-01-04 00:00:00	182	...	3.0
1	2015-01-04 01:00:00	138	...	3.0
2	2015-01-04 02:00:00	134	...	3.0
3	2015-01-04 03:00:00	72	...	3.0
4	2015-01-04 04:00:00	47	...	3.0

As informações de interesse aqui para realizar as previsões são as colunas `timestamp`, que contém as datas e os respectivos horários, e a coluna `cnt`, que traz a informação do número de bicicletas

compartilhadas naquele período. Com base nessas informações criamos o DataFrame `df` para realizar as previsões:

```
df = pd.DataFrame()
df['ds'] = dados_bikes['timestamp']
df['y'] = dados_bikes['cnt']
```

A princípio vamos realizar a previsão sem nenhuma alteração:

```
modelo = Prophet()
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataFramefuturo)
```

Então, vamos checar através do gráfico das componentes quais são as sazonalidades que o Prophet identificou:

```
modelo.plot_components(previsao, figsize=(10,9));
```

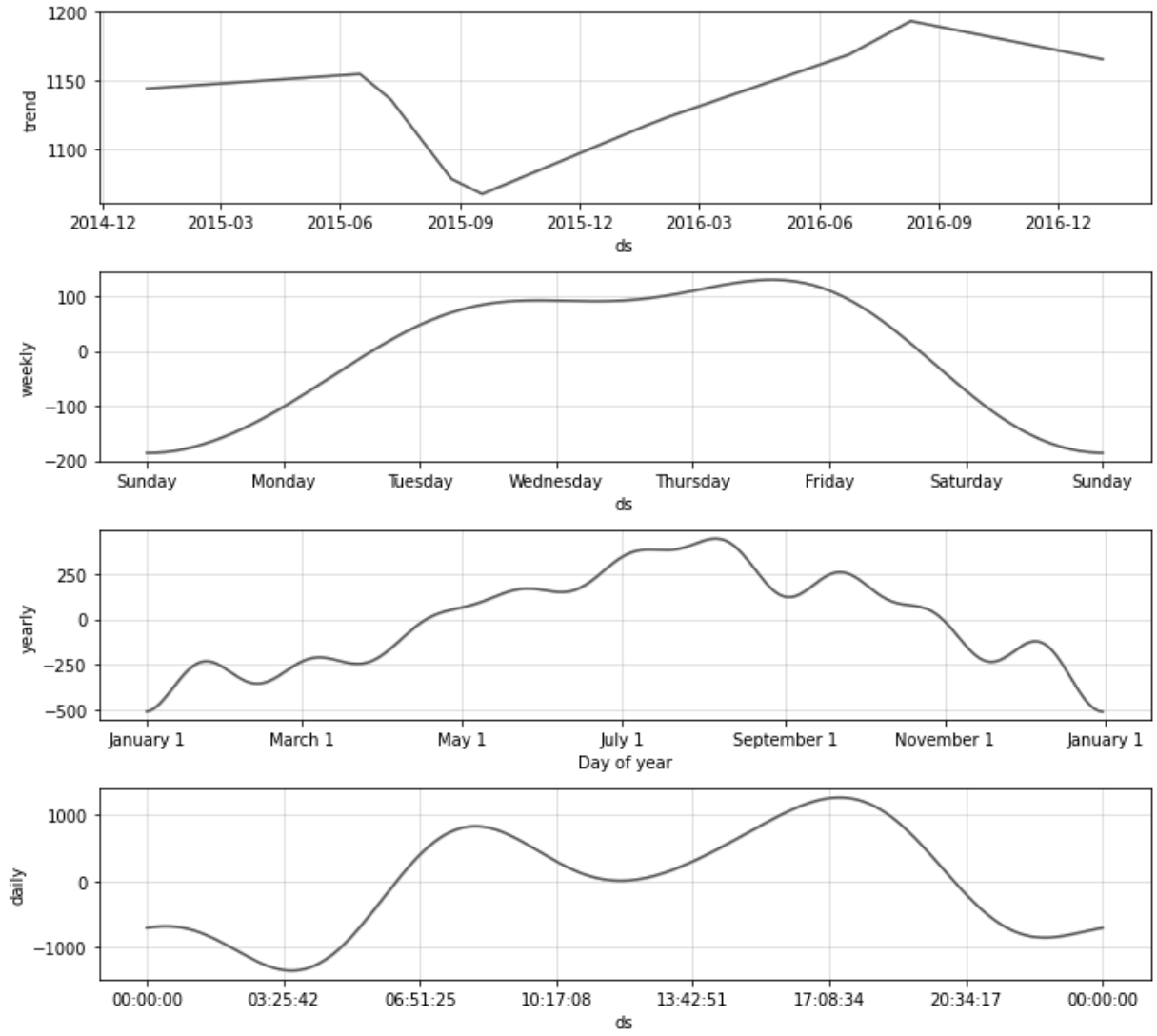

Figura 4.10: Gráfico das componentes identificados automaticamente pelo Prophet.

Nesse gráfico temos as sazonalidades semanal (*weekly*), anual (*yearly*) e diária (*daily*). É interessante utilizar aquela métrica do erro quadrático médio para ter uma ideia de como nosso ajuste está performando.

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'])
```

O erro quadrático médio é igual a 481544.86. A seguir vamos mostrar alguns testes sobre desativação e ativação das sazonalidades.

Para começar, vamos testar a desativação das sazonalidades semanal e diária com os argumentos `weekly_seasonality=False` e `daily_seasonality=False`. Em seguida, podemos adicionar a sazonalidade mensal com `add_seasonality`, que recebe um nome, o período em dias e o valor da Série de Fourier:

```
modelo = Prophet(weekly_seasonality=False, daily_seasonality=False)
modelo.add_seasonality(name='monthly', period=30.5, fourier_order=5)
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataFramefuturo)
```

Analisando o gráfico das componentes:

```
modelo.plot_components(previsao, figsize=(10,9));
```

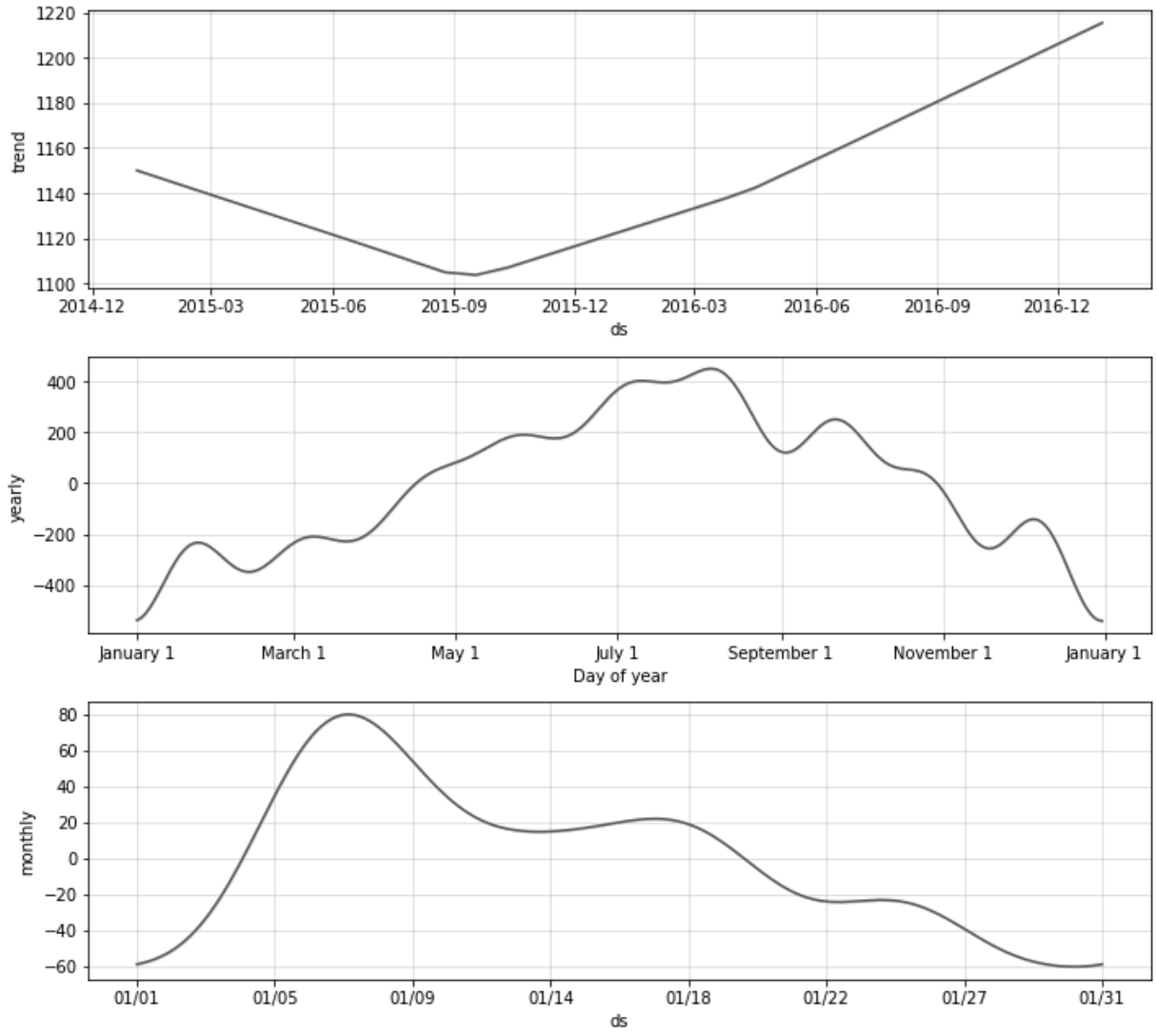

Figura 4.11: Gráfico das componentes desativando as sazonalidades semanal e diária e adicionando a sazonalidade mensal.

Temos apenas a sazonalidade anual e mensal agora! Observe como a linha de tendência mudou. Como estamos trabalhando com um *dataset* que contém os horários de contagens por dia, é bem provável que usar apenas as sazonalidades anual e mensal não seja a maneira mais adequada de realizar essa previsão. Isso pode ser confirmado com a métrica do erro quadrático médio:

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'])
```

A saída desse comando será um valor igual a 1104745.43, lembrando que, quanto mais próxima de zero, melhor essa métrica está. Nesse caso, ela aumentou consideravelmente.

Nosso segundo teste será mostrar o que ocorre quando adicionamos manualmente a sazonalidade diária, mas desativamos a semanal. Para sazonalidade diária utilizamos o período de 1 dia e Série de Fourier pode ser menor:

```
modelo = Prophet(weekly_seasonality=False)
modelo.add_seasonality(name='daily', period=1, fourier_order=3)
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataFramefuturo)
```

Analisando o gráfico das componentes:

```
modelo.plot_components(previsao, figsize=(10,9));
```

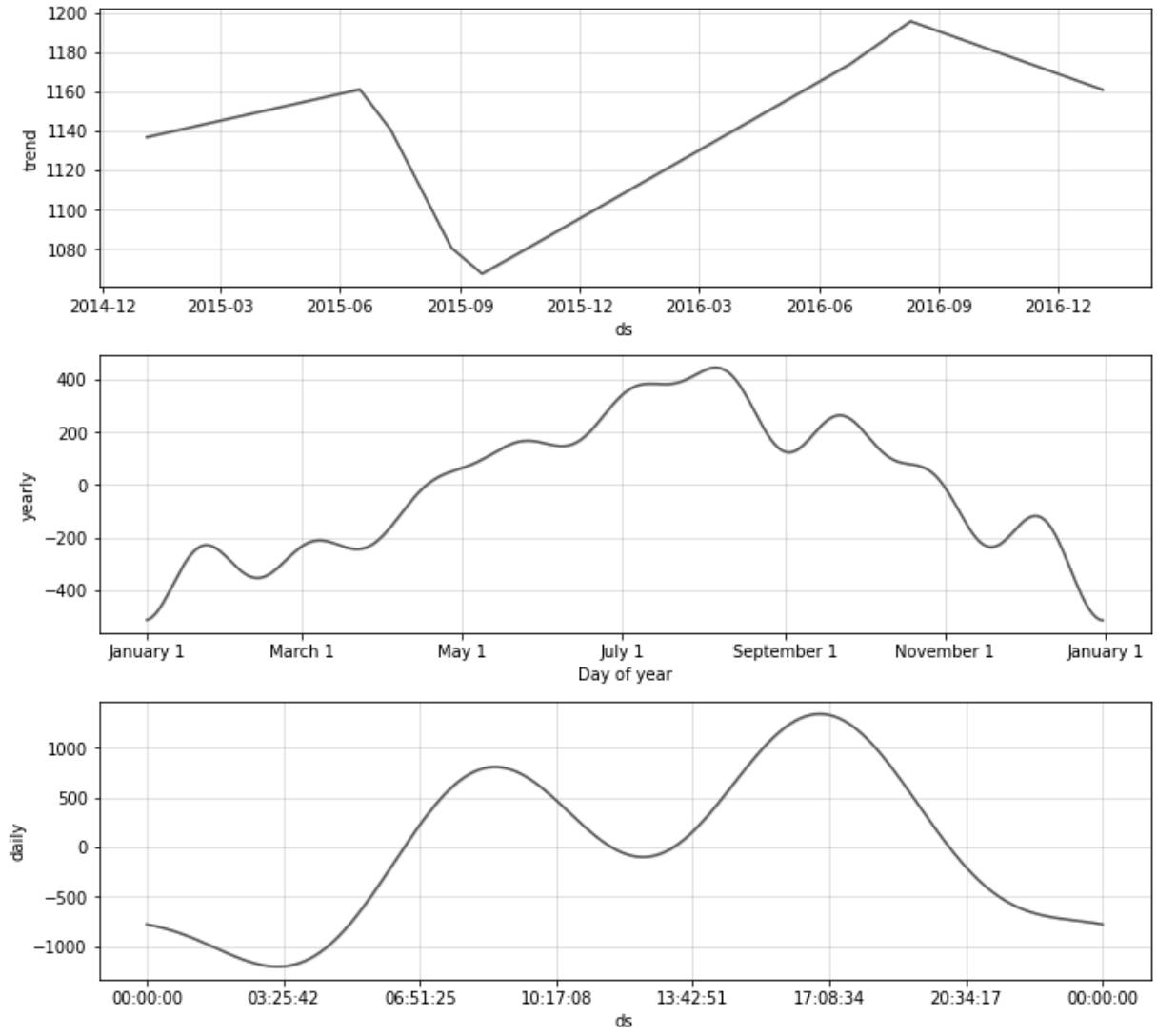

Figura 4.12: Gráfico das componentes desativando a sazonalidade semanal e adicionando manualmente a sazonalidade diária.

Então, temos apenas as sazonalidade anual e diária. Vamos checar a métrica:

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'])
```

O erro é igual a 509545.31. Será que é possível melhorar isso um pouco? Como discutimos ainda há pouco, seria interessante ter a sazonalidade horária incluída, já que as nossas contagens de bicicletas estão relacionadas aos horários.

Bom, vamos tentar o seguinte: desabilitar a sazonalidade semanal e adicionar a horária. Ao adicionar a sazonalidade horária, precisamos definir o `period=1/24`, pois estamos considerando um dia dividido por 24 horas:

```
modelo = Prophet(weekly_seasonality=False)
modelo.add_seasonality(name='hourly', period=1/24, fourier_order=3)
modelo.fit(df)
dataFramefuturo = modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataFramefuturo)
```

Analisando o gráfico das componentes:

```
modelo.plot_components(previsao, figsize=(10,9));
```

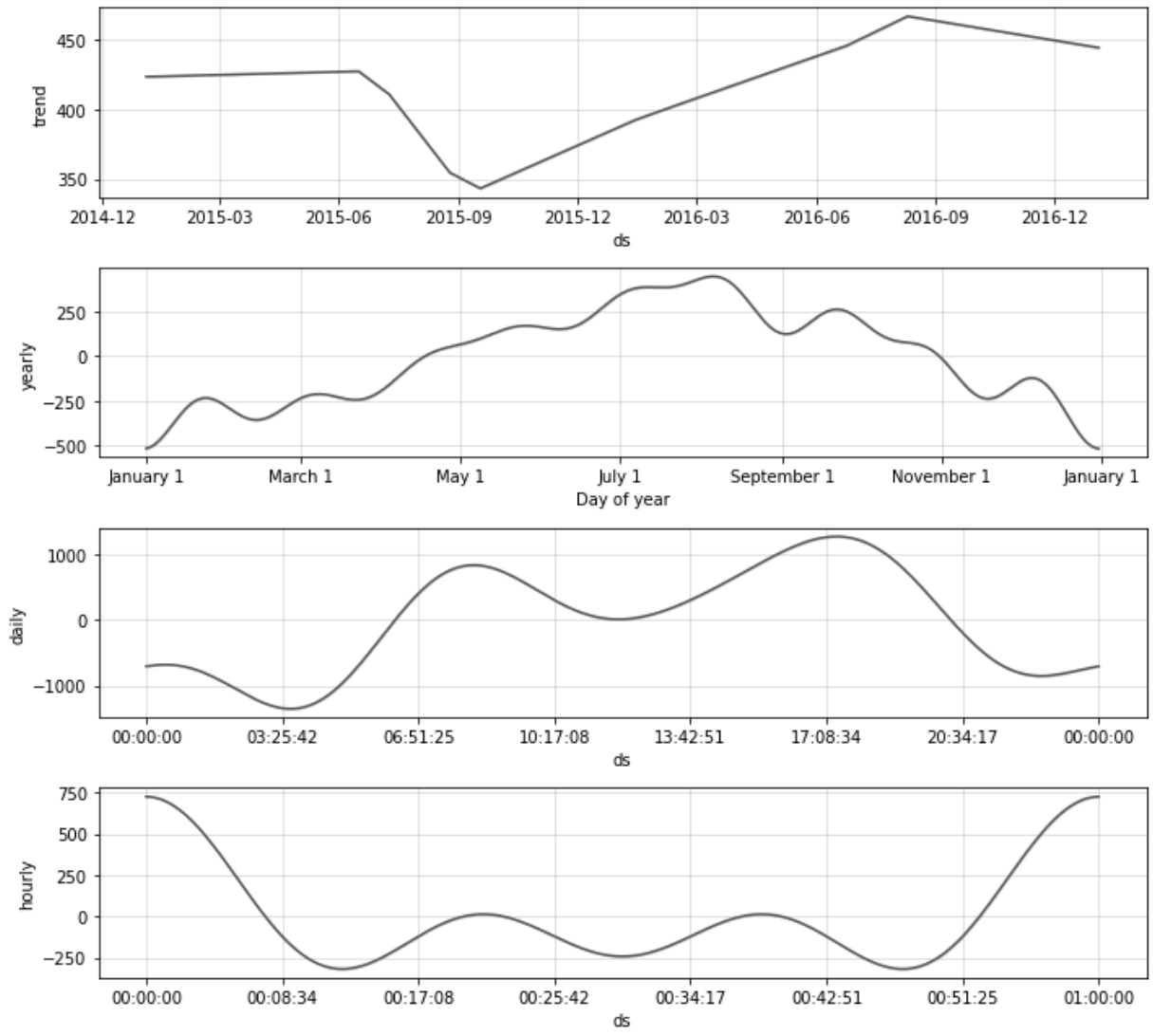

Figura 4.13: Gráfico das componentes desativando as sazonalidades semanal e adicionando manualmente a sazonalidade horária.

Por fim, temos as sazonalidades anual, diária e horária. Podemos avaliar a métrica:

```
mean_squared_error(y_true = df['y'], y_pred=previsao['yhat'])
```

Dessa vez, o erro quadrático médio é igual a 493744.45, ficando próximo ao erro das sazonalidades que o próprio Prophet adicionou automaticamente.

Neste capítulo, nós discutimos a diferença entre a sazonalidade aditiva e multiplicativa. Além disso, discutimos a Série de Fourier e notamos como alterar a Série impacta no modelo. Também, foi possível entender como especificar sazonalidades personalizadas, fazendo a desativação e ativação das sazonalidades desejadas.

No próximo capítulo, nós vamos discutir sobre o tratamento de *outliers*. *Outliers* são os pontos que estejam significativamente distantes dos outros pontos de dados ao longo um ou vários eixos diferentes. Isso será muito importante para tentar melhorar as previsões.

CAPÍTULO 5
Outliers

Geralmente, *datasets* têm valores discrepantes e essa série temporal não é exceção. Então, chegou o momento de finalizar a nossa previsão sobre as temperaturas da superfície da Terra fazendo uma etapa muito importante: o tratamento de *outliers*.

Outliers são pontos de dados que estejam significativamente distantes dos outros pontos de dados ao longo de um ou vários eixos diferentes. Esses pontos discrepantes podem ser dados incorretos, resultantes de um sensor mal calibrado, produzindo dados inválidos ou até mesmo um deslize do dedo no teclado durante a entrada de dados. É importante ressaltar que essas medições atípicas influenciarão qualquer aprendizado estatístico e, portanto, corrigir discrepâncias é uma etapa fundamental.

O Prophet é geralmente robusto a valores discrepantes leves. Porém, quando temos valores extremos, há dois problemas que o Prophet pode enfrentar: um problema com a sazonalidade e outro com intervalos de incerteza. Nós já falamos sobre sazonalidade no capítulo anterior e ao final deste capítulo também vamos discutir sobre esses intervalos de incerteza.

Para tratar os *outliers*, normalmente, precisamos removê-los quando sabemos que eles são fruto de algum erro das medições. A seguir vamos entender como *outliers* podem ser removidos do nosso *dataset*.

5.1 Tratamento de outliers

Antes de iniciar vamos carregar novamente o nosso DataFrame dos dados de treino:

```
df_treino.head()
```

index	ds	y
77110	1832-01-01 00:00:00	24.935
77111	1832-02-01 00:00:00	24.505
77112	1832-03-01 00:00:00	24.617
77113	1832-04-01 00:00:00	23.99
77114	1832-05-01 00:00:00	23.124

A nossa primeira etapa será resetar o index, para facilitar o trabalho com os dados, pois, como nós fizemos uma *query* no início para obter apenas os dados do Brasil, o index não está começando no número zero. Então, nós criamos uma nova variável: `novo_y`, que receberá o `df` sem o index:

```
novo_y = df_treino.reset_index()
novo_y.head(3)
```

	index	ds	y
0	77110	1832-01-01 00:00:00	24.935
1	77111	1832-02-01 00:00:00	24.505
2	77112	1832-03-01 00:00:00	24.617

Como resultado, temos o `novo_y`, com o index devidamente ordenado. Ter um índice inteiro, começando em zero, facilita o trabalho quando queremos comparar uma das colunas com nossa previsão, por exemplo.

Agora é o momento de remover os *outliers*. No capítulo 2 nós discutimos sobre a decomposição da previsão. Você deve se lembrar que nós tínhamos os valores do `yhat`, que são os valores da nossa previsão, e tínhamos também o `yhat lower` e o `yhat upper`. Se nós

queremos remover os *outliers* é necessário retirar apenas o que está acima do `yhat upper` e abaixo do `yhat lower`, pois o `yhat lower` é o limite inferior do intervalo de incerteza, e o `yhat upper` é o limite superior do intervalo de incerteza em torno da previsão final.

Sendo assim, o nosso critério para a remoção desses *outliers* será removê-los quando estiverem fora daquela região cinza clara em torno da curva principal, que foi realmente ajustada aos nossos dados.

Para fazer essa seleção, vamos criar a variável `sem_outliers` e digitar o seguinte comando:

```
sem_outliers = novo_y[(novo_y['y']> previsao['yhat_lower'][:2000]) &
(novo_y['y'] < previsao['yhat_upper'][:2000])]
sem_outliers.head(3)
```

	index	ds	y
0	77110	1832-01-01 00:00:00	24.935
1	77111	1832-02-01 00:00:00	24.505
2	77112	1832-03-01 00:00:00	24.617

Agora nós temos um DataFrame sem os *outliers* que estavam abaixo ou acima dos intervalos de confiança. Tanto é que agora temos 1627 linhas em vez de 2000. Será que conseguimos resultados melhores no nosso modelo sem os *outliers*? Vamos tentar! Criaremos um novo DaraFrame que será igual a `sem_outliers[['ds','y']]`:

```
df_sem_outliers = sem_outliers[['ds','y']]
```

Então, temos um DataFrame com os *outliers* removidos:

```
df_sem_outliers.head()
```

index	ds	y
0	1832-01-01 00:00:00	24.935

index	ds	y
1	1832-02-01 00:00:00	24.505
2	1832-03-01 00:00:00	24.617
3	1832-04-01 00:00:00	23.99
4	1832-05-01 00:00:00	23.124

Vamos rodar o modelo sem os *outliers*. Vamos considerar que o `periods` é igual a 0, para focarmos somente nos nossos dados de treino:

```
modelo = Prophet()
modelo.fit(df_sem_outliers)
dataframefuturo = modelo.make_future_dataframe(periods=0)
previsao = modelo.predict(dataframefuturo)
```

Plotando o gráfico:

```
modelo.plot(previsao, figsize=(18,6));
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

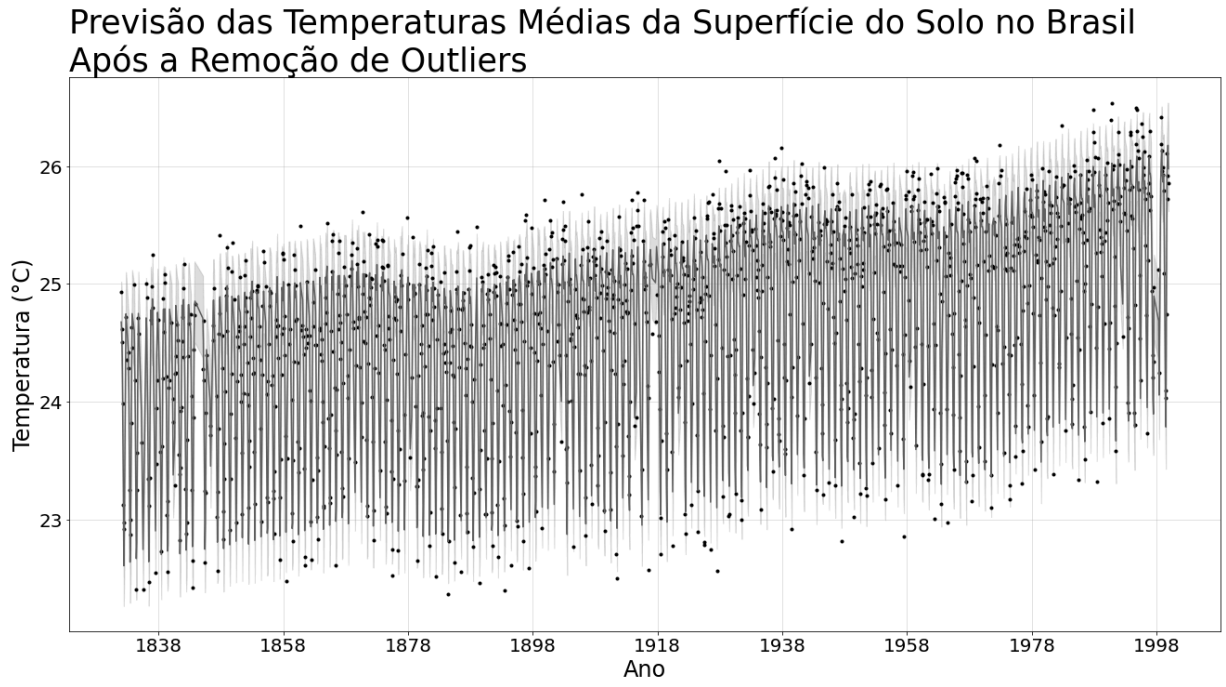

Figura 5.1: Gráfico de previsão após a retirada dos outliers.

A situação agora está bem melhor! Nós removemos praticamente todos os *outliers* que estavam fora do intervalo de confiança. Essa tarefa é muito importante, pois os *outliers* podem acabar enviesando negativamente as análises.

Nós podemos checar novamente aquela métrica de erro quadrático médio para confirmar se o nosso resultado melhorou:

```
mean_squared_error(y_true = df_sem_outliers['y'], y_pred=previsao['yhat']
[:1627])
```

Como resultado, temos um erro quadrático médio de apenas 0.076. Então, agora o ajuste dos dados realmente teve uma melhora! Essa é praticamente a nossa previsão final e agora nós vamos analisar e tentar interpretar um pouco mais os resultados. Chegou o momento de falar um pouco sobre os intervalos de confiança.

5.2 Intervalos de confiança

O intervalo de confiança é onde estão nossas possíveis soluções. O valor *default* do Prophet através do parâmetro `interval_width` é de 0.80, ou seja, de 80%. Utilizando esse valor *default* para um período de, por exemplo, 300 dias, temos o seguinte resultado:

```
modelo.plot(previsao, figsize=(18,9));
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.title('Previsão das Temperaturas Médias da Superfície do Solo no
Brasil \nIntervalo de confiança de 80%', loc = 'left', fontsize = 35)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

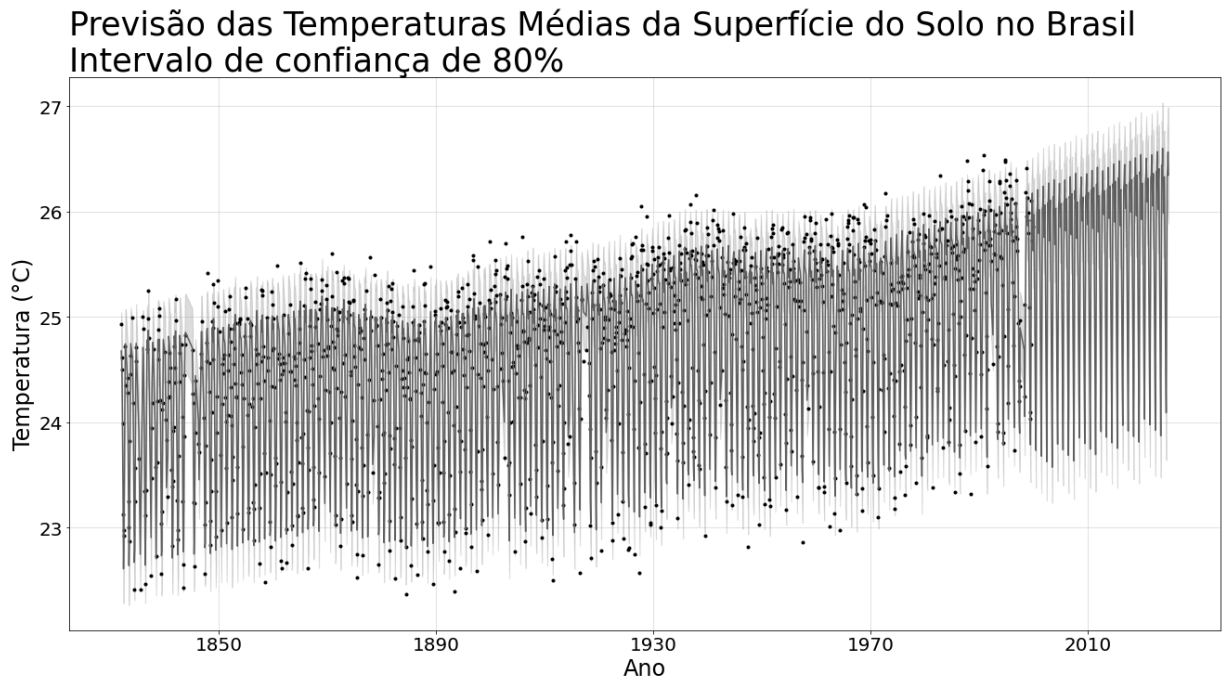

Figura 5.2: Gráfico de previsão de 300 dias após a retirada dos outliers, com intervalo de confiança de 80%.

A região que não contém os pontos em preto é a nossa previsão e a região mais clara em torno é o nosso intervalo de confiança, mas vamos analisar as componentes:

```
modelo.plot_components(previsao, figsize=(10,6));
```

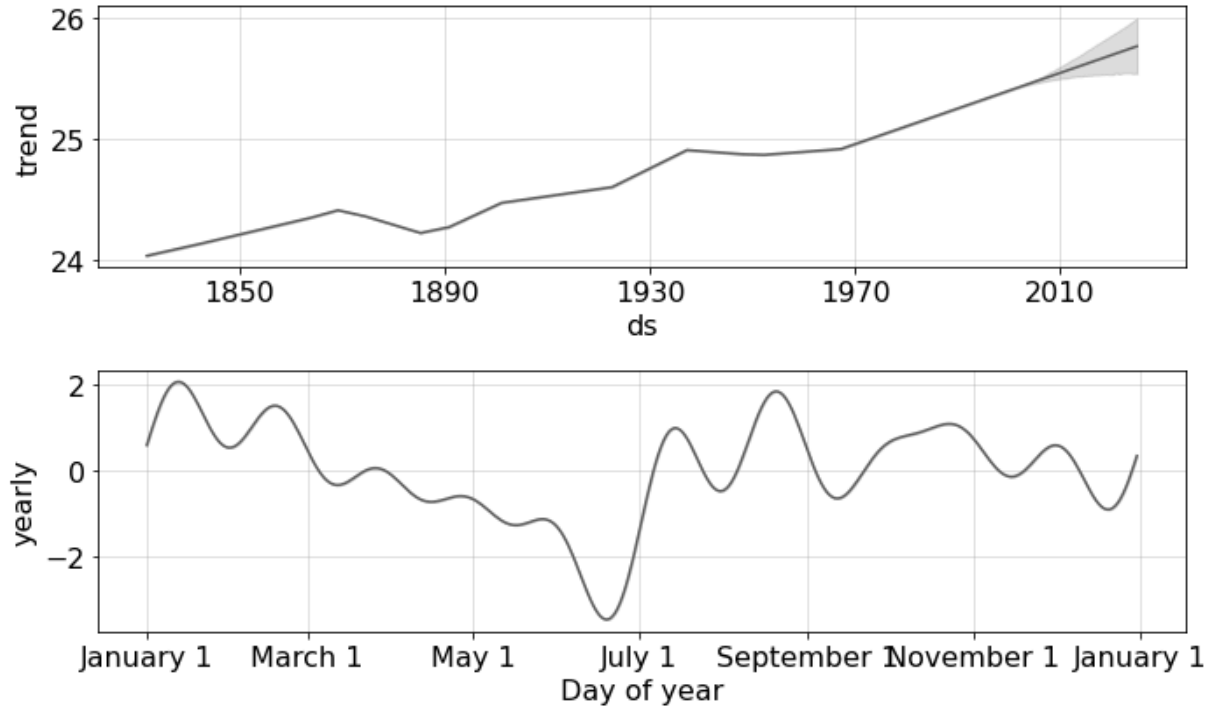

Figura 5.3: Gráfico das componentes da previsão de 300 dias após a retirada dos outliers, com intervalo de confiança de 80%.

No gráfico superior da tendência (*trend*), temos a faixa em cinza claro ao final da linha que corresponde ao intervalo de confiança. Para alterar o intervalo de confiança, podemos usar o argumento `interval_width` . Vamos testar alterar o intervalo para 0.95:

```
modelo = Prophet( interval_width=0.95)
modelo.fit(df_sem_outliers)
dataframefuturo = modelo.make_future_dataframe(periods=300, freq = 'M')
previsao = modelo.predict(dataframefuturo)
```

Plotando o gráfico:

```
modelo.plot(previsao, figsize=(18,9));
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.title('Previsão das Temperaturas Médias da Superfície do Solo no
Brasil \nIntervalo de confiança de 95%', loc = 'left', fontsize = 35)
```

```
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

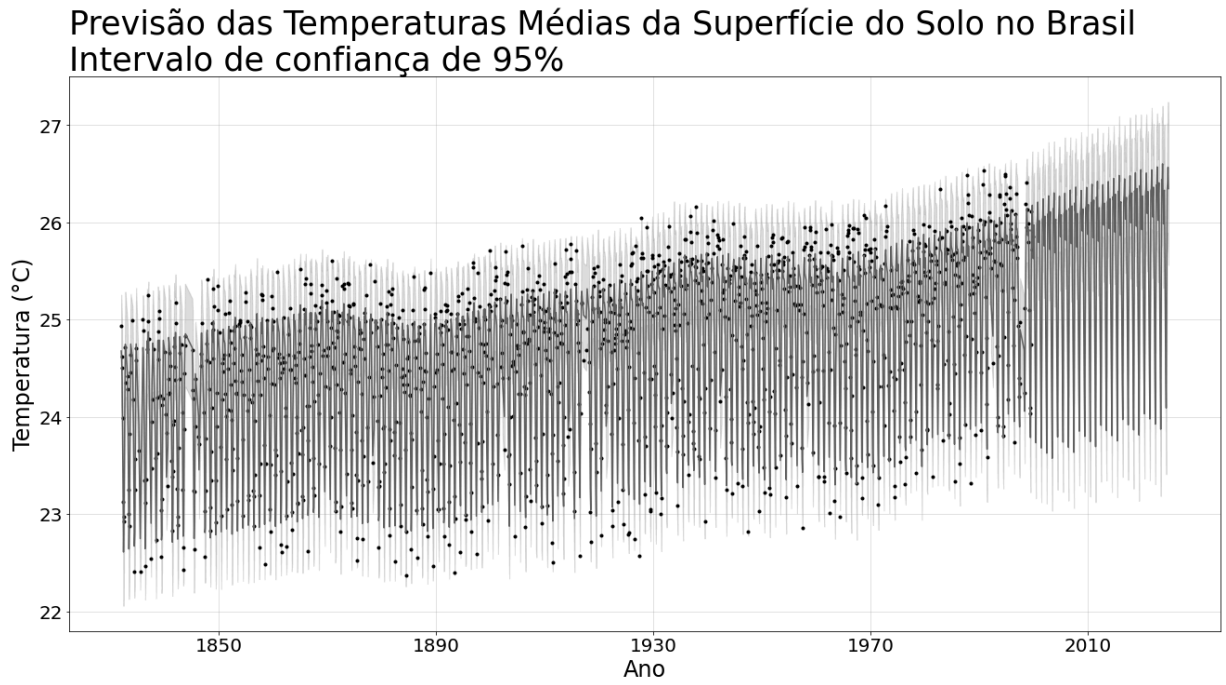

Figura 5.4: Gráfico de previsão de 300 dias após a retirada dos outliers, com intervalo de confiança de 95%.

Para entender o efeito do aumento do intervalo de confiança vamos checar as componentes, especificamente o componente da tendência (*trend*):

```
modelo.plot_components(previsao);
```

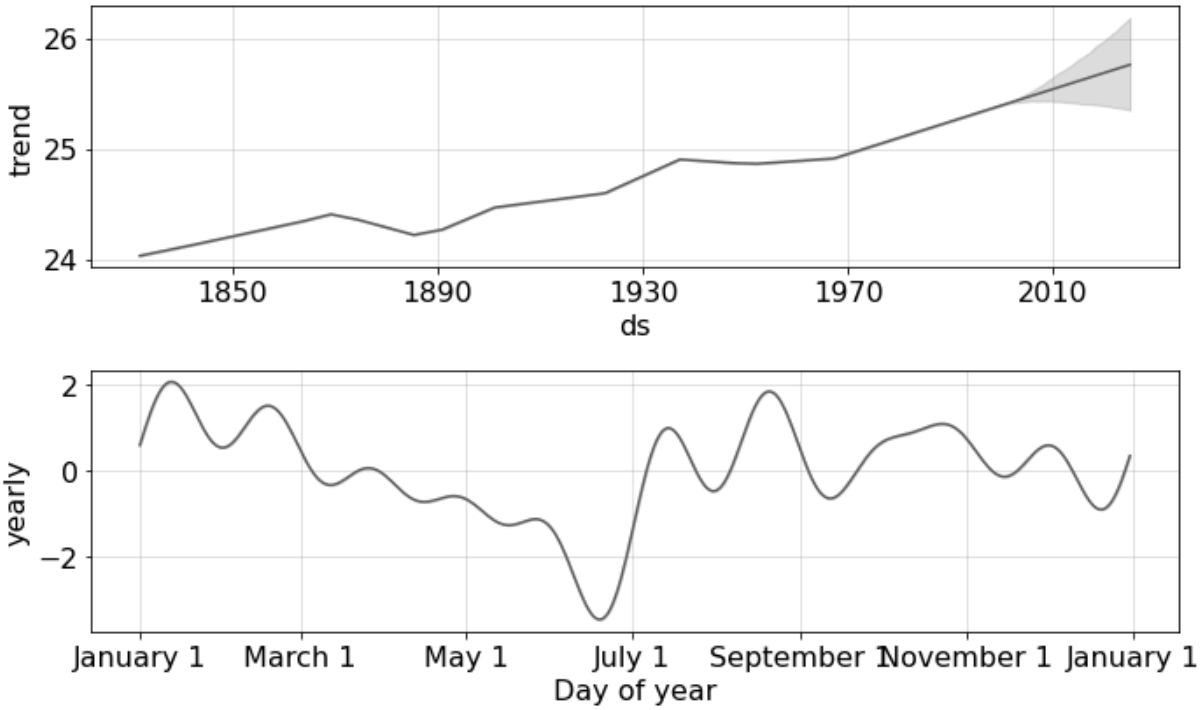

Figura 5.5: Gráfico das componentes da previsão de 300 dias após a retirada dos outliers, com intervalo de confiança de 90%.

Como podemos notar aqui, quando alteramos o intervalo de confiança, isso influencia no gráfico das componentes. Veja como a região em cinza claro, que representa o intervalo de confiança, ficou mais ampla.

O Prophet atribui esse intervalo de confiança para possíveis erros de ajuste em nossa série temporal. Portanto, o intervalo de confiança mostra as possíveis soluções que o Prophet está gerando.

Nós podemos analisar isso detalhadamente da seguinte forma: vamos gerar algo que chamaremos de amostras que vai receber o `modelo.predictive_samples` , que por sua vez receberá a previsão.

```
amostras = modelo.predictive_samples(previsao)
```

Em seguida, é possível *plotar* um gráfico para que seja possível visualizar o que são essas amostras. Então, faremos o `plt.plot` recebendo `previsao` e `amostras` :

```
plt.plot(previsao['ds'],amostras['trend']);
```

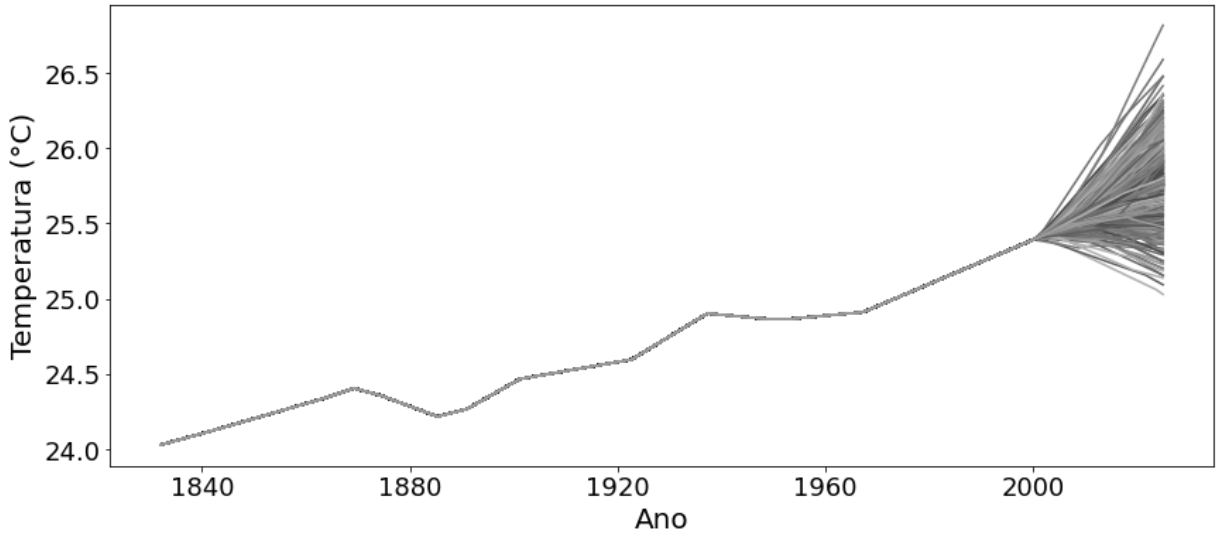

Figura 5.6: Gráfico das amostras.

O Prophet faz um ajuste parecido com o que fazemos em ajustes de uma reta. Você se lembra quando precisamos estimar coeficientes A e B de uma reta? Então, o Prophet "chuta" esses coeficientes e, depois do chute, ele ajusta os coeficientes até chegar a um bom resultado do ajuste aos dados. Nesse processo, o Prophet gera muitas respostas, que estão representadas pelas linhas dentro do intervalo de confiança no gráfico gerado. Essa é a ideia principal de um intervalo de confiança: um intervalo onde estão nossas possíveis soluções! Como pode ser observado aqui, a tendência pode mudar para cima ou para baixo dentro de um intervalo.

Bom, nós conseguimos melhorar o nosso modelo após a remoção dos *outliers*. No próximo capítulo nós vamos avaliar essa previsão com métricas, através de um processo bastante importante para saber como nossa previsão está performando: o *cross-validation*.

CAPÍTULO 6
Validação e métricas de performance

No capítulo anterior nós realizamos a retirada dos *outliers* e discutimos sobre os intervalos de confiança e como interpretá-los. Agora chegou um momento importante na previsão: validar o nosso resultado!

6.1 Validação cruzada

Você já deve ter notado que o conceito de separar dados em treino e teste é algo fundamental quando estamos trabalhando com aprendizado de máquina. Porém, se não tivermos muitas amostras ou muitas datas de exemplo nos dados de teste podemos ter bons resultados nesses dados apenas devido ao acaso. Portanto, queremos testar mais o nosso o modelo para estimar a sua performance em dados que não foram vistos durante o treino. A *cross-validation* ou validação cruzada é uma técnica que é usada para contornar esse problema.

Para realizar essa validação vamos importar o `cross_validation` :

```
from fbprophet.diagnostics import cross_validation
```

O Prophet inclui uma funcionalidade para validação cruzada de séries temporais, com o objetivo de medir o erro de previsão. Isso é feito selecionando pontos de corte (*cutoffs*) e, para cada um deles, é ajustado o modelo usando dados somente até aquele ponto de corte. Podemos então comparar os valores previstos com os valores reais.

A figura a seguir ilustra uma previsão onde o modelo se ajustou a um histórico inicial de 5 anos, e uma previsão foi feita no horizonte

de um ano.

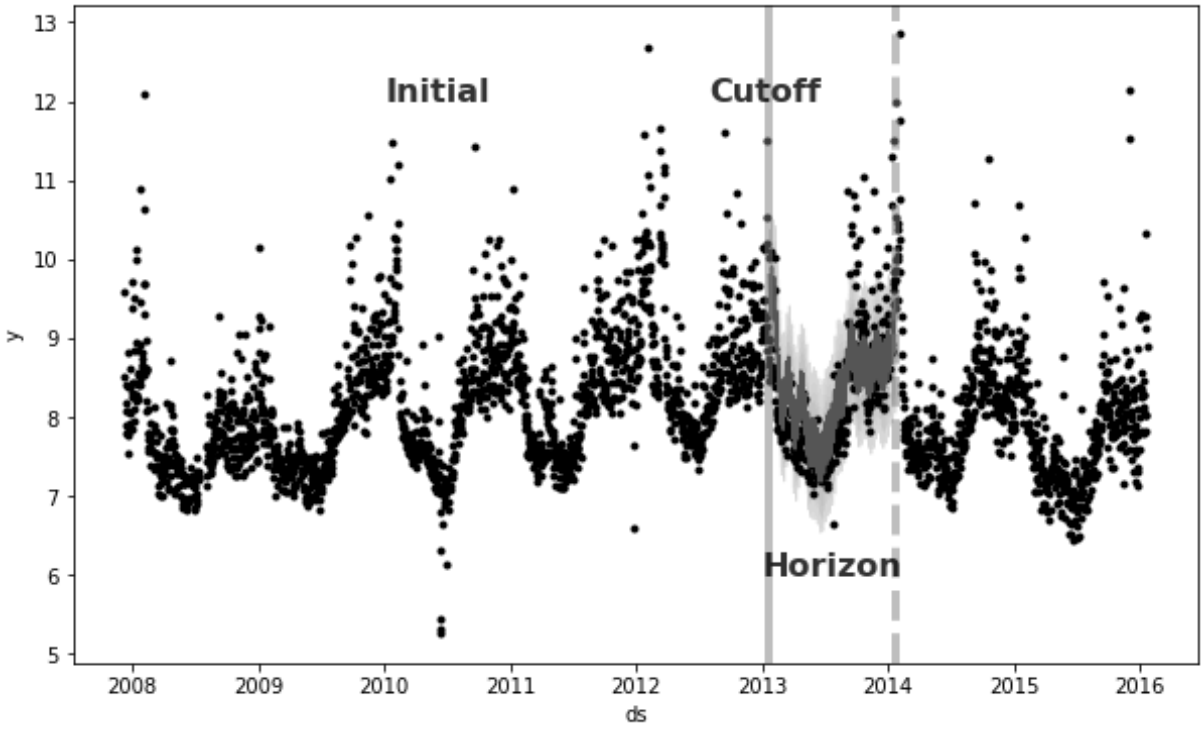

Figura 6.1: Gráfico representando o processo de validação cruzada utilizada no Prophet. Fonte: Documentação do Prophet, disponível em https://facebook.github.io/prophet/docs/diagnostics.html.

Na figura, temos a região inicial chamada de `initial`, que corresponde ao primeiro período de treinamento. Em seguida, temos o `cutoff`: o ponto de corte, ou seja, é a data em que cada `horizon` começará. Para cada `cutoff`, o modelo é treinado em todos os dados até esse `cutoff` e, em seguida, uma previsão será feita para o período do `horizon`.

Essa previsão é comparada com os valores conhecidos e avaliados. Em seguida, o modelo é treinado novamente em todos os dados até o segundo `cutoff` e o processo será repetido. A avaliação final de desempenho é a média do desempenho em cada `cutoff`.

Além disso, temos outro termo que não está apresentado neste gráfico, que é o `period`, a quantidade de tempo de cada `cutoff`.

Pode ser maior que o `horizon` ou menor que ele, ou até mesmo igual a ele.

Para realizar a *cross-validation*, temos que criar um DataFrame de validação cruzada, que chamaremos de `df_cv`. Essa variável, receberá o `cross_validation()`. Então, passamos o modelo, que foi obtido na nossa última previsão e, além dele, temos que passar o `initial`. Como nosso *dataset* é longo e tem muitos anos, vamos definir um `initial` de 20 anos (`initial='7300 days'`), lembrando que esse `initial` se refere à primeira quantidade de pontos que estamos utilizando para treino dos dados.

Outros parâmetros que precisam ser definidos são o `period` e o `horizon`. Vamos deixar o `period` com 5 anos e o `horizon` com 10 anos. Então: `period='1825 days'` e `horizon='3650 days'`. Depois que começarmos a rodar o `cross_validation`, ele vai fazer uma série de testes de dias à frente na série temporal.

Nosso treinamento vai começar com um período inicial de 20 anos. A nossa previsão será avaliada em um intervalo de 10 anos e, como definimos um período de 5 anos, o nosso modelo será treinado e reavaliado a cada 5 anos.

Vamos definir todos esses parâmetros a seguir:

```
df_cv = cross_validation(modelo, initial='7300 days', period='1865 days',
horizon = '3650 days')
```

Essa validação resulta em um total de 28 previsões. Esse número de previsões vai depender da quantidade de anos do nosso *dataset*. Em seguida, para entender ainda melhor como essa validação funciona, vamos definir o que chamaremos de `janela`. Em um primeiro momento, chamaremos essa janela de `janela = 1`. Depois faremos também o *cutoff*, que pegará o `df_cv` e uma coluna que chamamos de `cutoff` e que surgiu dentro do `df_cv`. Primeiramente, vamos visualizar as colunas e as linhas do `df_cv`:

```
df_cv
```

index	ds	yhat	yhat _lower	yhat _upper	y	cutoff
0	1855-03-01	24.79	24.50	25.09	24.98	1855-01-05
1	1855-06-01	22.98	22.68	23.28	23.18	1855-01-05
2	1855-07-01	23.22	22.91	23.52	23.14	1855-01-05
3	1855-08-01	24.23	23.94	24.52	24.17	1855-01-05
...	...	...	...	...	...	...
2786	1999-09-01	25.69	25.34	26.03	26.10	1989-12-03
2787	1999-10-01	25.84	25.49	26.23	25.92	1989-12-03
2788	1999-11-01	25.97	25.61	26.35	25.72	1989-12-03
2789	1999-12-01	25.75	25.39	26.11	25.85	1989-12-03

No `df_cv` encontramos o `yhat`, `yhat_lower`, `yhat_upper` e o `cutoff`. Portanto, os 7300 dias iniciais foram treinados, e depois do dia 1989-12-03, os dados foram testados.

Para entender melhor como o Prophet está testando os dados, um gráfico será gerado a seguir. Esse gráfico não precisa ser gerado quando estamos desenvolvendo um projeto, então vamos apenas mostrá-lo para melhorar ainda mais o entendimento sobre o processo.

Para isso, dentro do `df_cv` , vamos pegar a coluna de `cutoff` e indicar que queremos visualizar os valores únicos:

```
df_cv['cutoff'].unique()
```

Como saída, temos um array contendo as datas de `cutoff` . Como imprimimos os valores únicos, temos uma série de "cutoffs" ou janelas. Início de janela de dados:

```
array(['1855-01-05T00:00:00.000000000', '1860-01-
04T00:00:00.000000000','1865-01-02T00:00:00.000000000', '1870-01-
01T00:00:00.000000000','1874-12-31T00:00:00.000000000', '1879-12-
30T00:00:00.000000000','1884-12-28T00:00:00.000000000', '1889-12-
27T00:00:00.000000000','1894-12-26T00:00:00.000000000', '1899-12-
25T00:00:00.000000000','1904-12-24T00:00:00.000000000', '1909-12-
23T00:00:00.000000000','1914-12-22T00:00:00.000000000', '1919-12-
21T00:00:00.000000000','1924-12-19T00:00:00.000000000', '1929-12-
18T00:00:00.000000000','1934-12-17T00:00:00.000000000', '1939-12-
16T00:00:00.000000000','1944-12-14T00:00:00.000000000', '1949-12-
13T00:00:00.000000000','1954-12-12T00:00:00.000000000', '1959-12-
11T00:00:00.000000000','1964-12-09T00:00:00.000000000', '1969-12-
08T00:00:00.000000000','1974-12-07T00:00:00.000000000', '1979-12-
06T00:00:00.000000000','1984-12-04T00:00:00.000000000', '1989-12-
03T00:00:00.000000000'],dtype='datetime64[ns]')
```

Tudo isso fica um pouco mais claro em formato gráfico. Para conseguir construir um gráfico, pegamos apenas uma das janelas, a primeira, que é igual a 1. Então, vamos criar a variável `janela` aqui:

```
janela = 1
```

Também é necessário definir um corte `df_cv_cut = df_cv['cutoff']` . Deste `cutoff` , interessam-nos os valores que são exatamente iguais ao `cutoff` da primeira janela que adotamos:

```
cutoff = df_cv['cutoff'].unique()[janela]
df_cv_cut = df_cv[df_cv['cutoff'].values == cutoff]
```

Para plotar um gráfico vamos utilizar o código a seguir:

```
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111)
ax.plot(modelo.history['ds'].values, modelo.history['y'], 'k.')
ax.plot(df_cv_cut['ds'].values, df_cv_cut['yhat'], ls='-', c='k')
ax.fill_between(df_cv_cut['ds'].values, df_cv_cut['yhat_lower'],
                df_cv_cut['yhat_upper'], color='gray',
                alpha=0.2)
ax.axvline(x=pd.to_datetime(cutoff), c='gray', lw=4, alpha=0.5)
ax.set_ylabel('y')
ax.set_xlabel('ds')
```

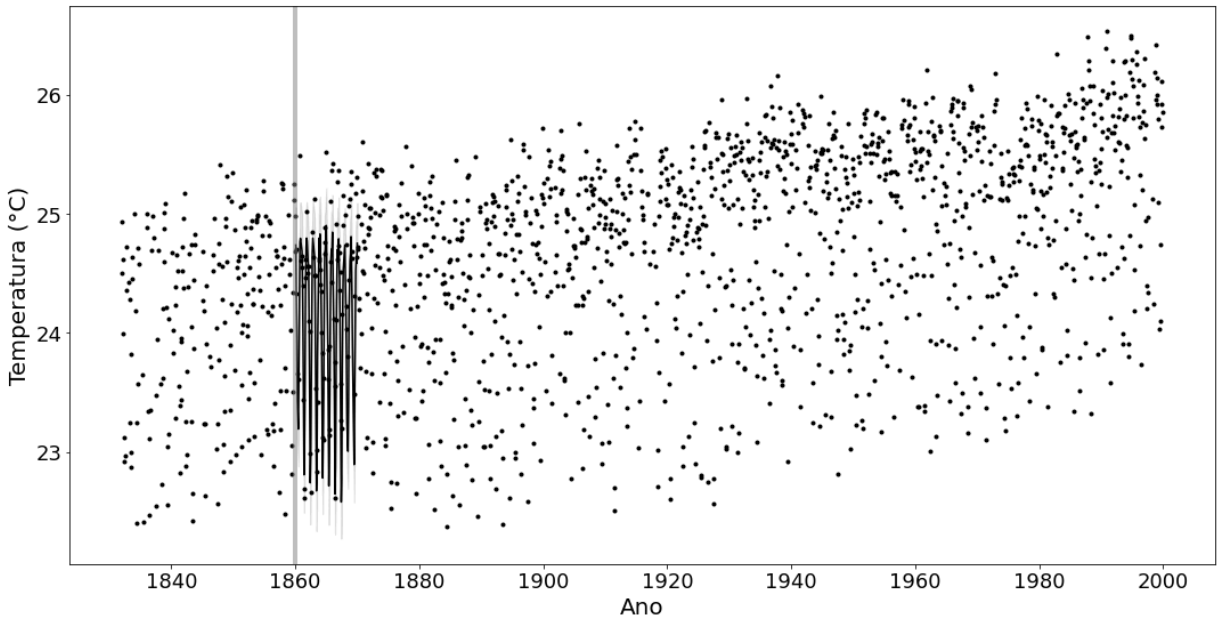

Figura 6.2: Gráfico da validação-cruzada, evidenciando a primeira janela.

A visualização deste gráfico tem por objetivo nos ajudar no entendimento do problema, não necessariamente a resolvê-lo melhor. Esse é o gráfico da série temporal, com os pontos pretos sendo os nossos dados de treino - valores de y. Mas, além disso, temos essa linha vertical em cinza claro, plotadas em um determinado ponto do gráfico (após o ano 1860). As linhas em preto se referem ao trecho que foi usado para teste. Então, o Prophet usou todos os pontos pretos anteriores à linha vertical cinza para treinar o modelo de série temporal e, depois, pegou esse trecho preto e utilizou para teste do modelo.

Podemos voltar ao código e fazer uma alteração na janela. Como temos muitos valores de `cutoff`, podemos alterar o valor para 10:

```
janela = 10
```

Agora, vamos rodar outra vez o `cutoff`:

```
cutoff = df_cv['cutoff'].unique()[janela]
df_cv_cut = df_cv[df_cv['cutoff'].values == cutoff]
```

E os comandos para gerar o gráfico:

```
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111)
ax.plot(modelo.history['ds'].values, modelo.history['y'], 'k.')
ax.plot(df_cv_cut['ds'].values, df_cv_cut['yhat'], ls='-', c=k')
ax.fill_between(df_cv_cut['ds'].values, df_cv_cut['yhat_lower'],
                df_cv_cut['yhat_upper'], color='gray',
                alpha=0.2)
ax.axvline(x=pd.to_datetime(cutoff), c='gray', lw=4, alpha=0.5)
ax.set_ylabel('y')
ax.set_xlabel('ds')
```

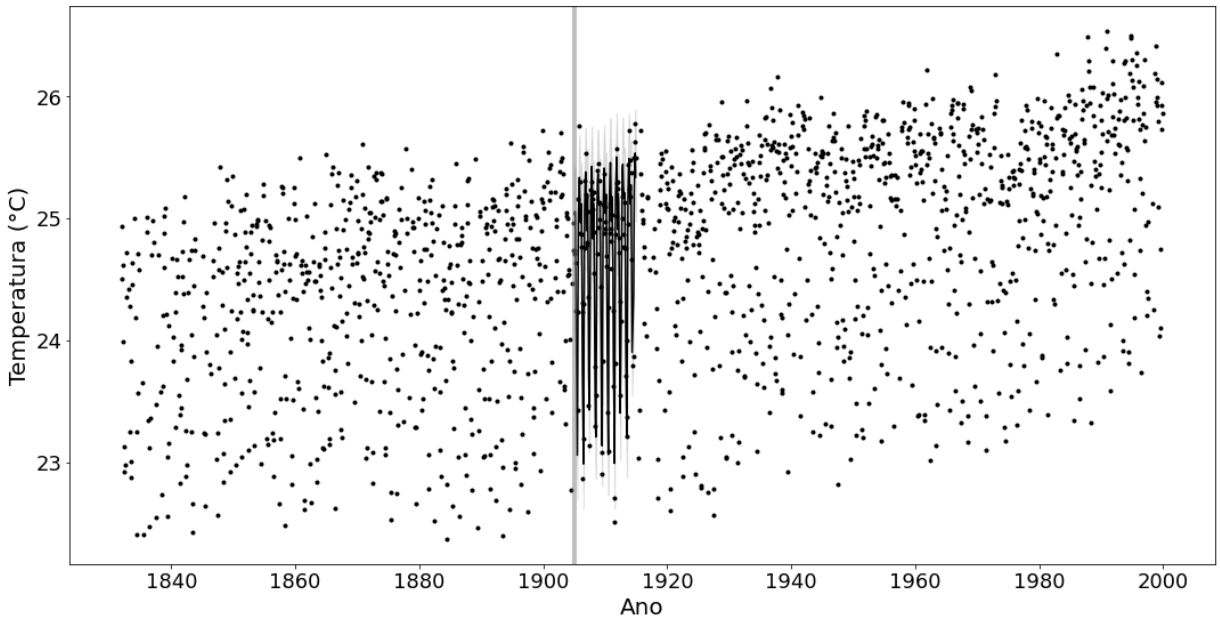

Figura 6.3: Gráfico da validação-cruzada, evidenciando a janela 10.

Ao mover a janela para a frente, para um valor maior, ele pegará todos os pontos anteriores à linha cinza do `cutoff`, e da janela para a frente, como uma janela de testes para validar os resultados.

Com isso, temos um bom exemplo do que acontece internamente quando fazemos o `cross_validation`. A seguir, vamos aplicar métricas de performance, para entender se o nosso modelo está bom ou não.

6.2 Métricas de Performance

Como você pode dizer que seu modelo é bom? Como você pode acreditar que as suas previsões para o futuro são confiáveis? A validação cruzada permitiu comparar previsões imparciais com valores reais. Agora chegou o momento de quantificar o desempenho do nosso modelo.

Para analisar as métricas de performance, precisamos fazer as seguintes importações:

```
from fbprophet.diagnostics import performance_metrics
from fbprophet.plot import plot_cross_validation_metric
```

Para analisar as métricas, vamos criar um DataFrame de performance, `df_p`, e passar para ele o `performance_metrics()`, que, por sua vez, receberá o `df_cv` que criamos na parte de validação cruzada.

```
df_p = performance_metrics(df_cv)
```

Vamos dar uma olhada nas cinco primeiras linhas do DataFrame de performance `df_p`:

```
df_p.head().T
```

index	0	1	2	3	4
horizon	361 days	362 days	363 days	364 days	365 days
mse	0.089	0.089	0.089	0.089	0.090
rmse	0.299	0.298	0.298	0.299	0.300
mae	0.250	0.249	0.249	0.250	0.251
mape	0.010	0.010	0.010	0.010	0.010
mdape	0.009	0.009	0.009	0.009	0.009
coverage	0.729	0.733	0.733	0.729	0.725

Aqui nós temos as métricas para cada `horizon`. Antes de tudo, vamos entender o que são cada uma dessas métricas:

MSE (*Mean Sqsared error* - Erro quadrático médio): É a soma da diferença quadrática entre cada valor de *y* real e o valor do *y* previsto. O MSE é a métrica de desempenho mais utilizada, mas também tem seu lado negativo: a unidade de MSE é o quadrado da sua unidade *y*, então não é tão simples de interpretar. Além disso, é sensível a *outliers*, embora isso possa ser desejável ou não, dependendo de seus dados e interpretação. Essa métrica nunca é negativa e quanto mais próxima de zero, melhor o modelo.

RMSE (*Root mean squared error* - Raiz do erro quadrático médio): É utilizada quando você quer dimensionar o MSE para as mesmas unidades dos seus dados, obtendo a raiz quadrada. As vantagens e desvantagens são as mesmas da MSE, mas suas unidades são mais interpretáveis.

MAE (*Mean absolute error* - Erro absoluto médio): Essa métrica é semelhante ao MSE, exceto pelo fato de calcular o valor absoluto de erro em vez de elevar ao quadrado. MAE pesa cada erro igualmente e, portanto, não coloca mais importância em *outliers*.

MAPE (*Mean absolute percent error* - Erro percentual absoluto médio): É outra métrica bastante comum, mesmo tendo baixa capacidade de representar o desempenho de um modelo. MAPE divide cada erro pelo valor do ponto de dados naquele erro e isso torna a métrica distorcida para representar os erros que ocorrem quando os valores são baixos. Outro problema é que, como você está dividindo por cada valor *y*, se algum dos valores de *y* for igual a zero, o cálculo produzirá um erro de divisão. Além disso, valores muito pequenos de y podem causar problemas de cálculo de ponto flutuante. O Prophet trabalha assim: ele vai detectar se algum valor de *y* é igual ou próximo de zero e, caso isso aconteça, ele simplesmente pulará os cálculos do MAPE e prosseguirá para as outras métricas exigidas.

MDAPE (*Median absolute percent error* - Erro percentual absoluto mediano): Essa métrica segue a mesma lógica da MAPE, porém usa a mediana em vez da média. Pode ser útil usar essa métrica com dados ruidosos, que apresentam muitos *outliers*. Imagine que você esteja trabalhando com dados que tenham grandes picos significativos. Nesse caso, utilizar a mediana seria mais capaz de suavizar as previsões.

Coverage (Cobertura): A cobertura é a porcentagem dos valores reais que se situam entre os limites de incerteza superior e inferior previstos. Por padrão, esses limites de incerteza cobrem 80% dos dados, então, seu valor de cobertura deve ser 0,8. A não ser que você altere aquele parâmetro `interval_width` discutido no capítulo anterior, quando abordamos os intervalos de confiança.

Mais informações sobre essas métricas podem ser encontradas em um artigo escrito pela Karoline Penteado, disponível em: https://www.alura.com.br/artigos/metricas-de-avaliacao-para-series-temporais. Nesse artigo também temos links para você explorar cada métrica na documentação do Scikit-Learn.

Vamos plotar o gráfico de uma dessas métricas, por exemplo, a métrica MSE, que é muito utilizada na validação de séries temporais:

```
plot_cross_validation_metric(df_cv,metric='mse');
```

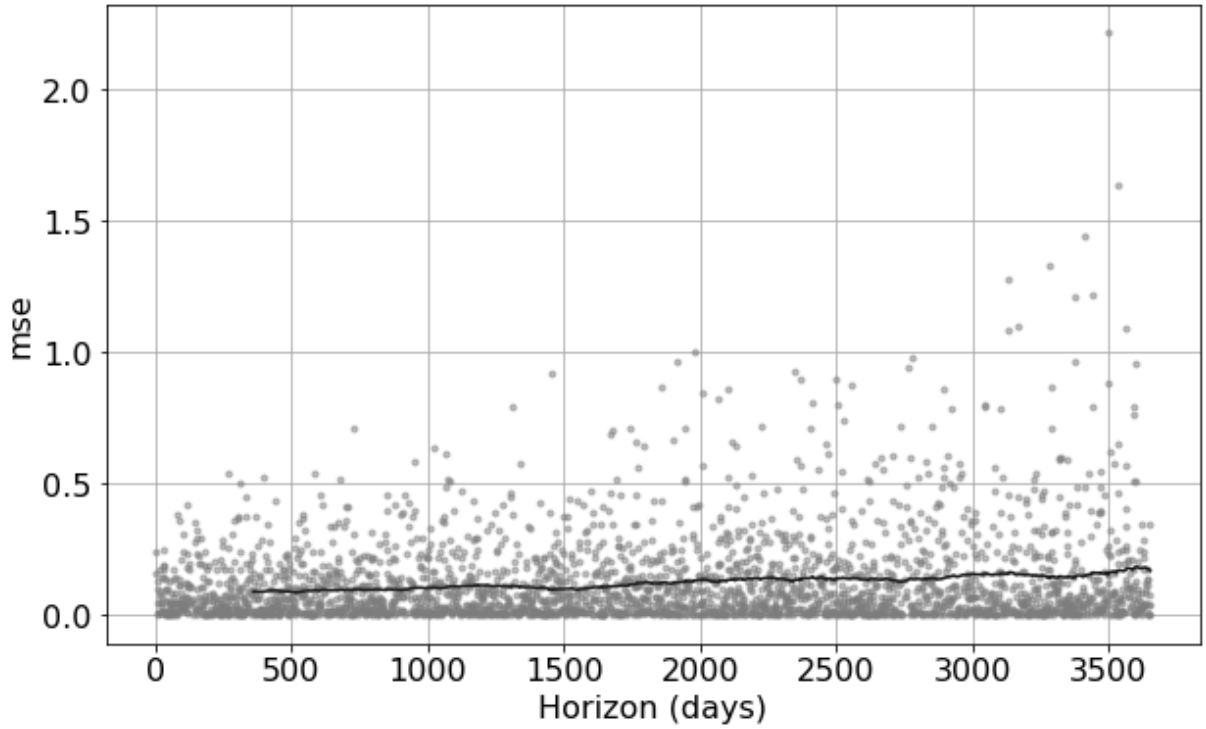

Figura 6.4: Gráfico da métrica de performance MSE (Mean squared error - Erro quadrático médio).

O gráfico mostra o MSE a cada horizonte, e é possível notar que, conforme os horizontes aumentam, maior é a dispersão dos pontos. Esses pontos em cinza claro são os erros em torno da previsão. Essa linha em cinza escuro corresponde à média dos valores e, quanto mais próximos dessa linha os pontos estiverem, melhor.

O ideal é que essa nuvem de pontos fique consistente a cada teste. A métrica MSE nunca gera valores negativos e, quanto mais próxima de zero, melhor é o modelo! Nesse caso, os valores de MSE não estão altos e a dispersão é um pouco mais alta apenas nos últimos horizontes, indicando que temos um bom modelo.

Nós terminamos as nossas análises, realizamos a validação cruzada e acabamos de entender como analisar as métricas de performance. Prophet se revelou uma biblioteca muito poderosa, já que ela

modela, treina, testa, identifica a sazonalidade automaticamente e ainda faz a validação cruzada.

No capítulo seguinte, vamos discutir sobre o ajuste de hiperparâmetros e sobre como descobrir quais são os ajustes mais adequados aos nossos dados.

CAPÍTULO 7
Hiperparâmetros

No Prophet é possível ajustar os hiperparâmetros de uma forma mais ágil em vez de ficar fazendo diversos testes até encontrar quais modificações nos parâmetros trazem as melhores métricas como resultado. Nós podemos fazer isso com o *Grid Search*, uma maneira prática de verificar exaustivamente todas as combinações razoáveis de hiperparâmetros contra um indicador de desempenho, que pode ser uma daquelas métricas que discutimos no capítulo anterior. Com isso, você escolhe a melhor combinação para treinar o seu modelo.

Como já foi possível observar no decorrer dos capítulos, o Prophet apresenta vários parâmetros de entrada que podem ser considerados como ajustes. A seguir temos algumas recomendações gerais para ajuste de hiperparâmetros que podem ser um bom ponto de partida.

7.1 Parâmetros que podem ser ajustados

De acordo com a documentação do Prophet os seguintes parâmetros podem ser ajustados:

- `changepoint_prior_scale` : é provável que este seja o parâmetro mais impactante, pois ele determina a flexibilidade da tendência e, em particular, o quanto a tendência muda nos *changepoints*. Se este parâmetro for um valor muito pequeno, a tendência será subajustada e, portanto, a variação que deveria ter sido modelada com uma mudança de tendência acaba sendo tratada com o termo de ruído. Enquanto isso, se o valor for muito alto, a tendência se ajustará demais e, no caso mais extremo, você pode acabar com a tendência capturando a sazonalidade anual.

O *default* é de 0.05 e funciona para muitas séries temporais, mas isso pode ser ajustado, sendo um intervalo de 0.001 a 0.5 o mais indicado.

- **`sazonality_prior_scale`** : é o parâmetro que controla a flexibilidade da sazonalidade. Então, da mesma forma, um valor alto permite que a sazonalidade se ajuste a grandes flutuações e valores baixos diminuem a magnitude da sazonalidade. O *default* é 10, que basicamente não aplica nenhuma regularização. Um intervalo razoável para ajuste provavelmente é entre 0.01 a 10. Lembre-se de que, quando definido para 0.01, a magnitude da sazonalidade é forçada a ser muito pequena.

- **`holiday_prior_scale`** : este parâmetro controla a flexibilidade para ajustar os efeitos de feriados. Semelhante ao `sazonality_prior_scale` , o *default* é 10, que basicamente não aplica nenhuma regularização, pois geralmente temos várias observações de feriados e podemos fazer um bom trabalho ao estimar seus efeitos. Também pode ser ajustado em um intervalo de 0.01 a 10, como a `sazonality_prior_scale` .

- **`sazonality_mode`** : temos as opções `additive` ou `multiplicative` . O *default*, como já comentamos em capítulos anteriores, é `additive` , mas muitas séries temporais terão sazonalidade multiplicativa. Isso é fácil de identificar apenas olhando para a série temporal e observando se a magnitude das flutuações sazonais cresce com a magnitude da série temporal. Inclusive, no capítulo 4, temos um exemplo que ilustra bem a diferença entre os tipos de sazonalidade.

7.2 Parâmetros que talvez podem ser ajustados

- **`changepoint_range`** : esta é a proporção da série temporal em que a tendência pode mudar. O *default* é 0.8, ou seja, 80% da série. E o que isso quer dizer? Que o modelo não se ajustará a nenhuma mudança de tendência nos últimos 20% da série temporal. Em algumas situações pode ser benéfico ser menos conservador e aumentar essa proporção. Contudo, seria muito difícil ajustar esse parâmetro efetivamente com validação cruzada sobre os *cutoffs*. A capacidade do modelo de generalizar a partir de uma mudança de tendência nos últimos 10% da série temporal, por exemplo, seria difícil de aprender observando os *cutoffs* anteriores que podem não ter mudanças de tendência nos últimos 10%. Portanto, certamente é melhor não ajustar esse parâmetro. Em casos específicos entre 0.8 a 0.95 pode ser um intervalo razoável.

7.3 Parâmetros que provavelmente não seriam ajustados

- **`growth`** : as opções disponíveis neste parâmetro são `linear` (linear) e `logistic` (logística), sendo `linear` o *default*. Isso provavelmente não será ajustado, pois, se houver um ponto de saturação conhecido e crescer até esse ponto, ele será incluído e a tendência logística será usada, caso contrário, a tendência será linear.

- **`changepoints`** : este parâmetro serve para especificar manualmente os locais dos *changepoints*.

- **`n_changepoints`** : este é o número de *changepoints* colocados automaticamente. O *default* é igual a 25 e costuma ser suficiente para capturar as mudanças de tendência em uma série temporal típica. Em vez de aumentar ou diminuir o número

de *changepoints*, provavelmente será mais eficaz focar em aumentar ou diminuir a flexibilidade nessas mudanças de tendência, o que é feito com `changepoint_prior_scale`.

- **`yearly_seasonality`**: por padrão, é `auto` e isso ativará a sazonalidade anual se houver um ano de dados e, caso contrário, desativará. As opções são `auto`, `True` e `False`. Se houver mais de um ano, em vez de tentar desativá-lo, possivelmente, será mais eficaz deixá-lo ativado e desativar os efeitos sazonais ajustando `sazonality_prior_scale`.

- **`weekly_seasonality`**: por padrão, é `auto` e isso ativará a sazonalidade semanal se houver uma semana de dados e, caso contrário, desativará.

- **`daily_seasonality`**: por padrão, é `auto` e isso ativará a sazonalidade diária se houver dados diários e, caso contrário, desativará.

- **`holidays`**: sua finalidade é passar um DataFrame de feriados especificados. Os efeitos de feriados podem ser ajustados com `holidays_prior_scale`.

- **`interval_width`**: a previsão do Prophet retorna intervalos de incerteza para cada componente, como `yhat_lower` e `yhat_upper` para a previsão `yhat`. Estes são calculados como quantis da distribuição preditiva posterior e o `interval_width` especifica quais quantis usar. O *default* de 0.8 fornece um intervalo de previsão de 80%. Você pode alterar isso para 0.95 se quiser um intervalo de 95%. Isso afeta apenas o intervalo de incerteza e não altera a previsão e, portanto, não precisa ser ajustado.

7.4 Ajustando hiperparâmetros com Grid Search

Para ajustar os hiperparâmetros nos nossos dados o primeiro passo será criar um dicionário com os possíveis parâmetros a serem testados. Parâmetros ideais para testes como esse são: `changepoint_prior_scale`, `seasonality_prior_scale` e `seasonality_mode`. Nesse caso, não vamos utilizar o `holiday_prior_scale`, pois não adicionamos feriados ao modelo. Quanto maior o `changepoint_prior_scale`, maior é a flexibilização da tendência e, como esse é o parâmetro mais impactante, vamos testar valores de 0.001 a 0.5. Já o parâmetro `seasonality_prior_scale` controla a flexibilidade da sazonalidade. Um valor grande permite que a sazonalidade se ajuste a grandes flutuações; um valor pequeno diminui a magnitude da sazonalidade. Então, testamos valores entre 0.01 a 10.

Por fim, o modo da sazonalidade ser aditivo ou multiplicativo vai depender das flutuações dentro da série de dados. Então, criamos o seguinte dicionário e armazenamos na variável `param_grid`:

```
param_grid = {'changepoint_prior_scale': [0.001, 0.01, 0.1, 0.5]
              'seasonality_prior_scale': [ 0.01, 0.1, 1.0, 10.0]
              'seasonality_mode': ['additive',
                                   'multiplicative']}
```

A seguir é necessário importar o `itertools`, para rodar as iterações dos parâmetros adicionados ao dicionário, pois teremos 32 combinações diferentes.

```
import itertools
```

Além de rodar essa iteração, será necessário criar uma lista com uma das métricas de performance. Como no capítulo anterior falamos muito sobre o MSE (erro quadrático médio), vamos usá-lo aqui também:

```
all_params = [dict(zip(param_grid.keys(), value))
              for value in itertools.product(
                                   *param_grid.values())]
mse_values= []
```

Em seguida, nós criamos o código a seguir para rodar a `cross-validation` para cada combinação de parâmetros. Então, passamos os parâmetros a serem analisados ao Prophet. Os dois asteriscos antes de `params` servem para pegar os nomes dos argumentos e com os nomes é possível trazer seus valores, já que os nomes são organizados em um dicionário. Na *cross-validation* utilizamos os mesmos parâmetros do capítulo anterior.

```
for params in all_params:
    modelo = Prophet( **params).fit(df_sem_outliers)
df_cv = cross_validation(modelo, initial='7300 days', period='1825 days',
horizon = '3650 days', parallel='processes')
df_p = performance_metrics(df_cv, rolling_window=1)
mse_values.append(df_p['mse'].values[0])
```

Dependendo da quantidade de parâmetros testados, esta etapa pode demorar! Para analisar os resultados, criamos um DataFrame do `all_params` e adicionamos uma coluna com os valores de MSE:

```
results = pd.DataFrame(all_params)
results['mse'] = mse_values
results
```

Aqui estão os nossos resultados! São 32 possibilidades e temos os valores de cada parâmetro. Na coluna final, temos os valores da métrica MSE:

	changepoint_prior_scale	seasonality_prior_scale	seasonality_mode	mse
0	0.500	10.00	additive	0.136306
1	0.500	10.00	multiplicative	0.135355
2	0.500	1.00	additive	0.136637
...	...	...	...	...
29	0.001	0.10	multiplicative	0.103240

	changepoint_prior_scale	seasonality_prior_scale	seasonality_mode	mse
30	0.001	0.01	additive	0.105030
31	0.001	0.01	multiplicative	0.102697

Como descobrir qual foi a melhor combinação? Para isso, criamos a variável `best_params` e pedimos para que seja mostrada a combinação que tenha o menor valor de MSE. Antes disso, importamos a biblioteca **Numpy**, pois vamos utilizar a função `argmin()` que retorna uma matriz de índices dos valores mínimos ao longo do eixo fornecido.

```
import numpy as np

best_params = all_params[np.argmin(mse_values)]
print(best_params)
```

O melhor resultado é o seguinte:

```
{'changepoint_prior_scale': 0.001, 'seasonality_prior_scale': 0.1,
'seasonality_mode': 'additive'}
```

Esta foi a nossa melhor combinação e chegou o momento de rodar a previsão para 300 meses com os `best_params` :

```
modelo = Prophet(changepoint_prior_scale=0.001,
seasonality_prior_scale=0.1, seasonality_mode='additive')
modelo.fit(df_sem_outliers)
dataframefuturo = modelo.make_future_dataframe(periods=300, freq = 'M')
previsao = modelo.predict(dataframefuturo)
```

Vamos checar como ficou o gráfico da nossa previsão?

```
modelo.plot(previsao, figsize=(18,9));
ax = fig.add_subplot()
plt.title('Previsão das Temperaturas Médias da Superfície do Solo no
Brasil \nApós Ajuste de Hiperparâmetros', loc = 'left', fontsize = 35)
plt.xticks(fontsize=20)
```

```
plt.yticks(fontsize=20)
plt.xlabel('Ano', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.show()
```

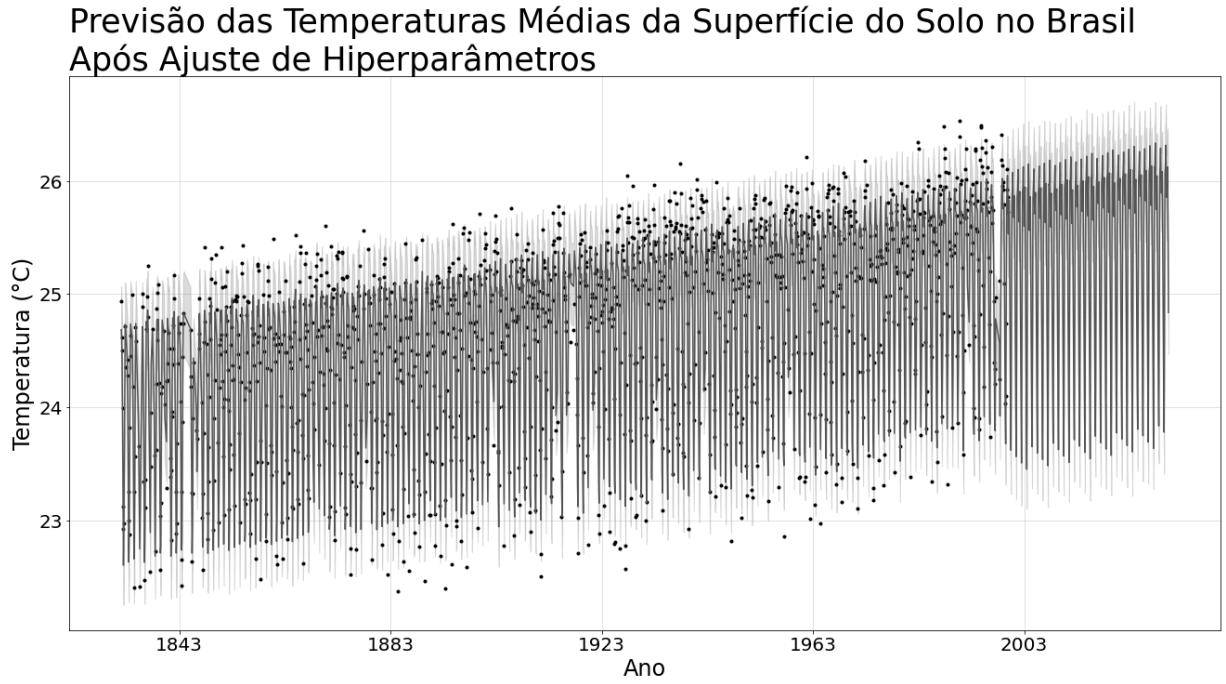

Figura 7.1: Ajuste aos dados após definição dos hiperparâmetros.

Como resultado temos esse gráfico com os dados de teste. Para finalizar este capítulo, que tal salvar o nosso modelo final?! Após salvo, o modelo pode ser atualizado e colocado em produção.

Para começar a salvar o modelo é necessário fazer importação do `json`. Json é um formato muito comum para troca de dados, por ser leve e simples:

```
import json
```

Também é necessário fazer duas importações a partir do `fbprophet.serialize`:

```
from fbprophet.serialize import model_to_json
from fbprophet.serialize import model_from_json
```

O `model_to_json` tem como função a exportação do modelo para o formato `.json`. Enquanto isso, o `model_from_json` permitirá o carregamento do arquivo após ter sido salvo.

Para salvar o modelo, vamos dar a ele o nome de `climate_change_br.json` e é preciso especificar o `w` para dizer que estamos escrevendo. Bastar utilizar o `json.dump` e passar o modelo final. Como estamos fazendo uma exportação utilizamos o `file_out`:

```
with open('climate_change_br.json', 'w') as file_out:
    json.dump(model_to_json(modelo), file_out)
```

Agora o modelo foi salvo e é possível fazer o download dele no Google Colab. Basta clicar no ícone de pasta (Arquivos) no menu esquerdo, que o arquivo salvo estará lá.

Para carregar o modelo salvo, digitamos `with open` e chamamos aquele nome dado ao arquivo. Porém, agora usamos `r` pois queremos ler o arquivo json. Na segunda linha criamos a variável modelo e ela será igual a `model_from_json`. Entre parênteses é passado `json.load(file_in)`:

```
with open('climate_change_br.json', 'r') as file_in:
    modelo = model_from_json(json.load(file_in))
```

Se você quiser ver o gráfico de previsão do modelo salvo, podemos plotá-lo da seguinte forma:

```
previsao = modelo.predict()
modelo.plot(previsao)
```

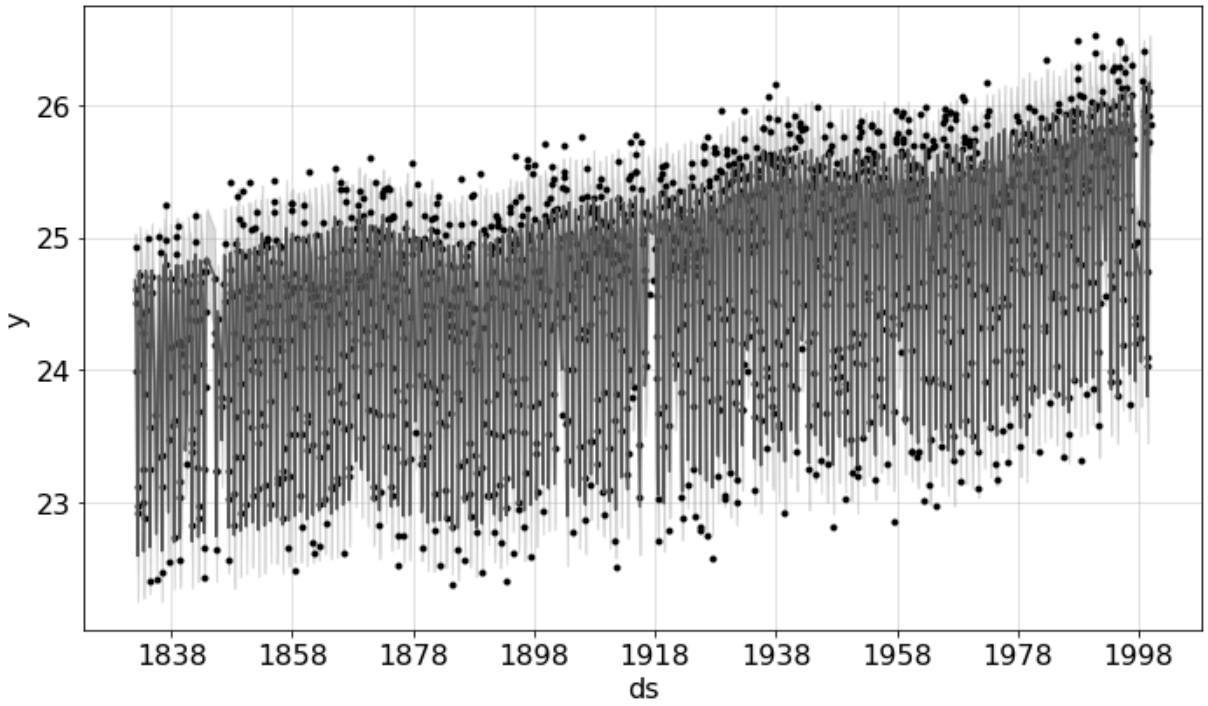

Figura 7.2: Modelo final após ajuste dos hiperparâmetros.

Agora temos o modelo salvo e aprendemos como carregá-lo. A partir daqui modificações podem ser feitas, pois uma situação comum com um modelo de previsão é a necessidade de reajustá-lo conforme mais dados chegam.

No próximo capítulo usaremos a biblioteca **Plotly** para construir alguns gráficos interativos.

CAPÍTULO 8
Gráficos interativos com Plotly

Neste capítulo, nós usaremos a biblioteca Plotly para construir alguns gráficos interativos das previsões. O Plotly é um pacote de visualização completamente separado do Matplotlib, que usamos ao longo dos capítulos anteriores.

E por que usar o Plotly? Imagine um gráfico interativo, que traz informações sobre os valores de previsão ao passar o mouse, permite aumentar e diminuir o zoom e possibilita a seleção de períodos específicos. Este capítulo foi escrito para lhe dar dicas de como usar essa ferramenta interativa.

Para utilizar esta biblioteca no Google Colab a primeira etapa será realizar as seguintes importações:

```
import plotly
import plotly.offline as py
import plotly.graph_objs as go
from plotly.offline import plot, iplot
import cufflinks as cf  # para conectar o plotly ao pandas
cf.go_offline()
plotly.offline.init_notebook_mode(connected=True)
import plotly.io as pio
pio.renderers.default = 'colab'
```

Agora, a partir do `fbprophet.plot` vamos fazer as importações a seguir:

```
from fbprophet.plot import (
    plot_plotly,
    plot_components_plotly,
    plot_seasonality_plotly
)
```

Com essas importações, conseguimos obter o gráfico de previsão, componentes e sazonalidade.

8.1 Gráfico interativo da previsão

Vamos iniciar *plotando* a previsão do nosso modelo. Para isso, utilizamos o `plot_plotly` e passamos o modelo e a previsão. A segunda linha de código, `py.iplot (fig)` , é análoga a `plt.show ()` de Matplotlib, e permite que o gráfico seja exibido.

```
fig = plot_plotly(modelo, previsao)
py.iplot(fig)
```

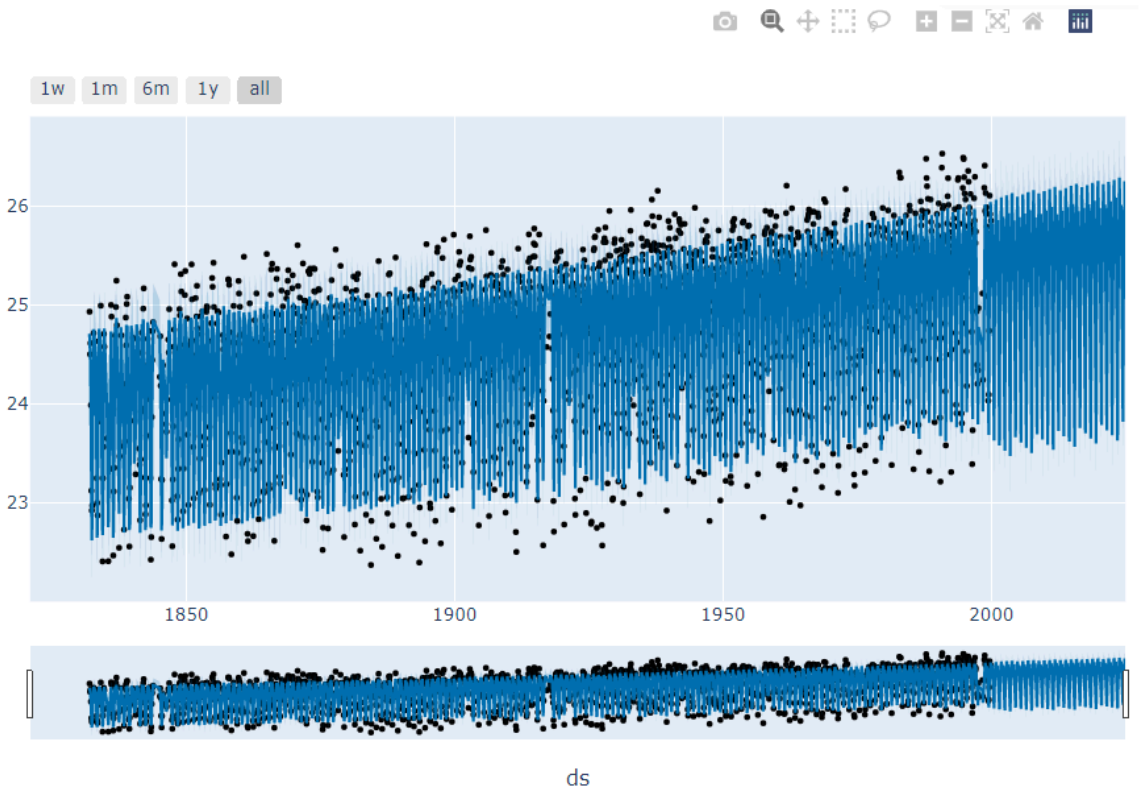

Figura 8.1: Gráfico interativo da previsão.

Note que a barra de ferramentas Plotly está no canto superior direito da figura. Com essa barra podemos:

- salvar a figura no formato png;
- ampliar áreas específicas;
- percorrer a figura;
- selecionar alguma região específica;
- aumentar ou diminuir o zoom.

Acima da figura, no canto esquerdo, temos períodos que podem ser explorados: uma semana *(1w)*, um mês *(1m)*, seis meses *(6m)*, um ano *(1y)* ou o período todo *(all)* .

Além disso, podemos incluir o argumento `trend = True` para adicionar a linha de tendência ao gráfico:

```
fig = plot_plotly(modelo, previsao, trend=True)
py.iplot(fig)
```

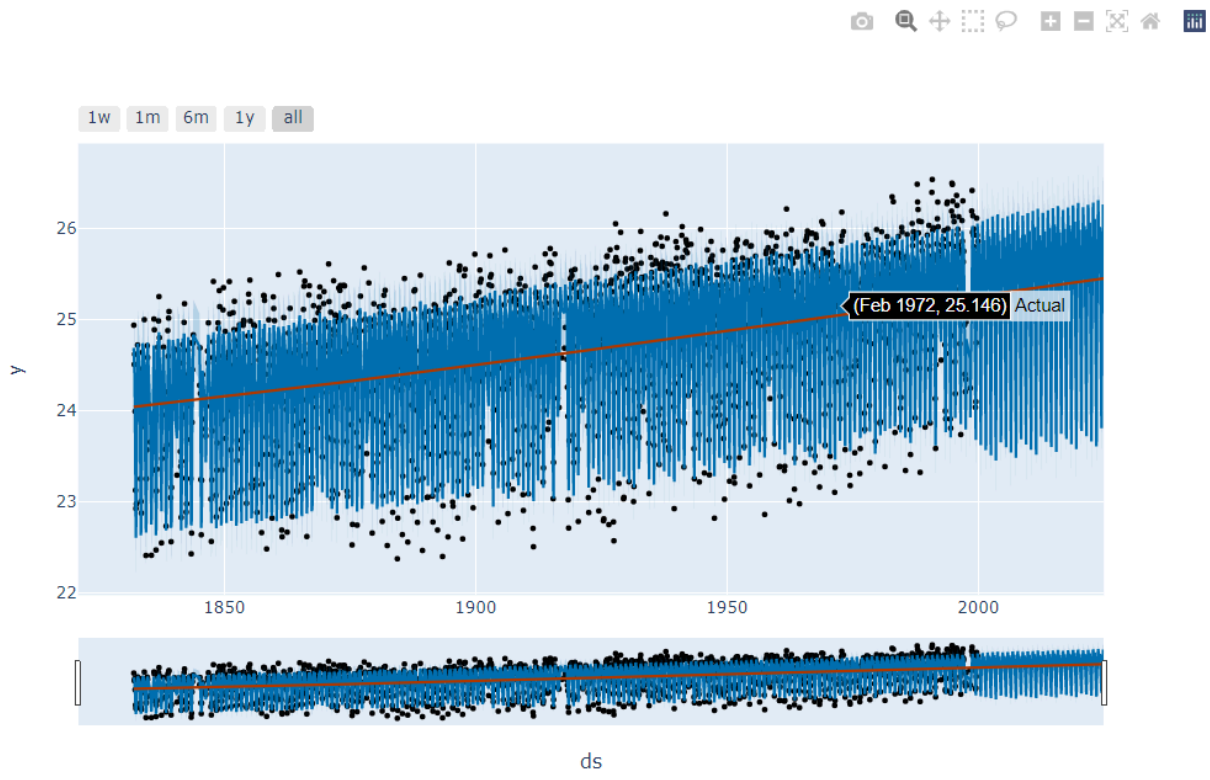

Figura 8.2: Gráfico interativo da previsão contendo a linha de tendência.

Observe que, quando passamos o *mouse* pelos dados, temos as datas e os respectivos valores das temperaturas. Inclusive, podemos ver os valores de `y` previstos se percorrermos a área sem os pontos, que corresponde ao período de previsão.

8.2 Gráfico interativo das componentes

A seguir, vamos obter o gráfico das componentes com o Plotly. É a mesma coisa que nós fizemos com o Matplotlib, mas agora inclui interatividade. Podemos incluir o argumento `figsize` para definir o tamanho da figura:

```
fig = plot_components_plotly(modelo, previsao, figsize=(800, 175))
py.iplot(fig)
```

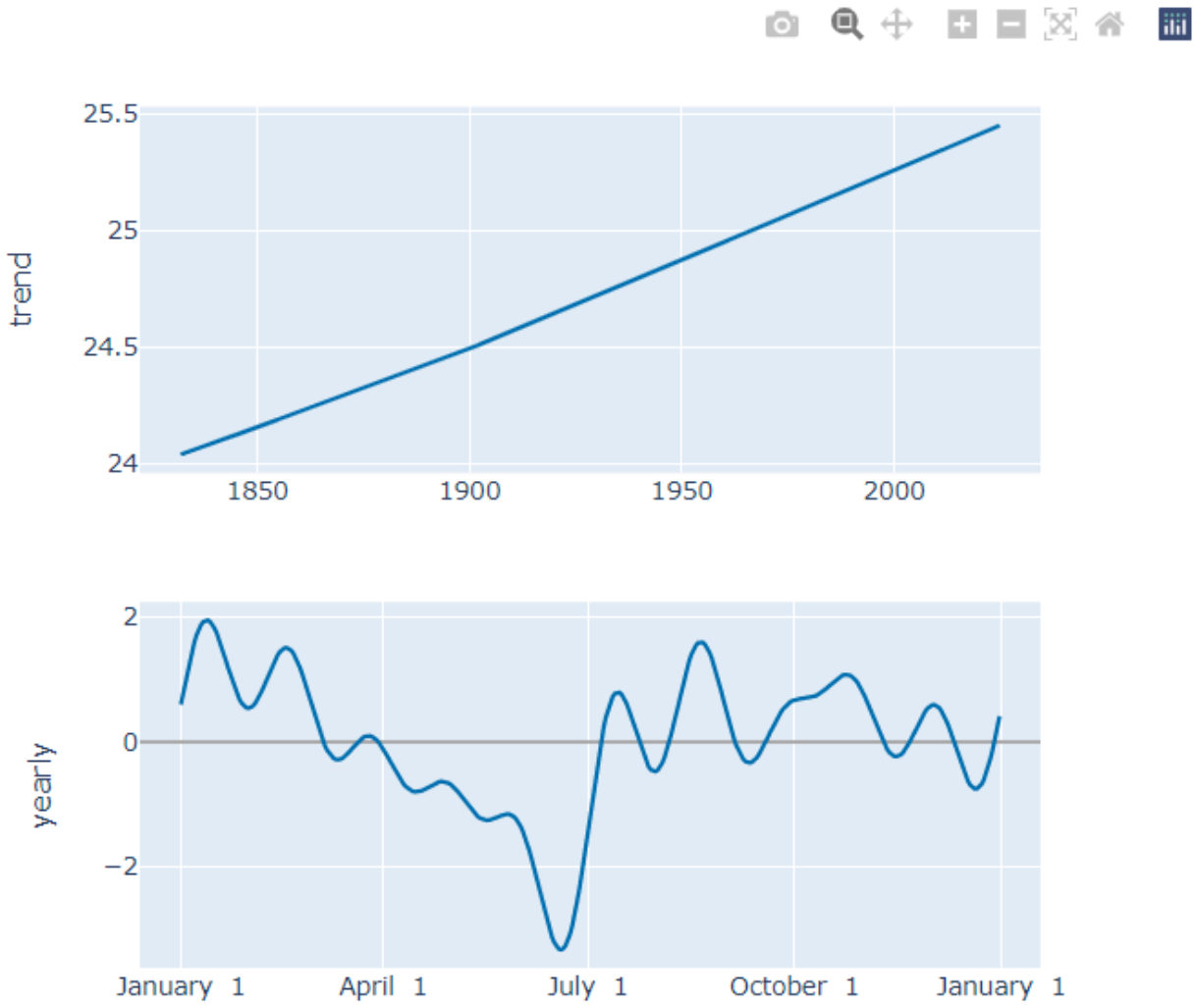

Figura 8.3: Gráfico interativo das componentes da previsão.

8.3 Gráfico interativo da sazonalidade

Para finalizar, vamos analisar somente a sazonalidade dos nossos dados utilizando a função `seasonality_plotly` :

```
fig = plot_seasonality_plotly(modelo, 'yearly')
py.iplot(fig)
```

Figura 8.4: Gráfico interativo da sazonalidade anual.

Nós aplicamos um novo formato de visualização aos nossos dados. Imagine que interessante mostrar seus resultados de previsão de séries temporais de forma interativa para o seu público-alvo! Certamente, chamará a atenção das pessoas que estão observando os seus resultados. Por isso nós encorajamos você a explorar esses gráficos em um notebook no Google Colab.

Bom, nós chegamos ao final das análises com o *dataset* de mudanças climáticas. Desde o primeiro capítulo, nós exploramos os dados, criamos nosso primeiro modelo e nos aprofundamos cada vez mais. Discutimos sobre sazonalidade e intervalos de confiança e, por fim, aplicamos a validação cruzada e entendemos como aplicar a métricas de performance.

Além disso, utilizamos o *Grid Search* para ajustar os hiperparâmetros e salvamos o modelo final, para que você possa compartilhá-lo ou abri-lo mais tarde sem exigir que o modelo seja retreinado.

Mas não acabamos por aqui! No capítulo seguinte temos um assunto muito importante quando falamos sobre séries temporais, que não discutimos ainda: o efeito de feriados! Como nosso *dataset* não era ideal para isso, nós pensamos em dados da área de finanças para explorar isso no próximo capítulo.

Capítulo 9
Modelagem de feriados e eventos especiais

Até o momento estudamos uma base de dados voltada para algo global. Nesse tipo de estudo existe uma certa dificuldade em correlacionar nossos dados com eventos que ocorrem no dia a dia. Por outro lado, se estudarmos as vendas de uma loja, com certeza, teremos a influência de certas datas, como é o exemplo da Black Friday. Por isso, neste capítulo vamos nos direcionar a um tipo de dado que atrai o interesse quando lidamos com séries temporais.

Vamos utilizar dados do mercado de ações. Estudar a variação do preço de uma ação nos permite verificar a correlação desses preços com datas específicas e também aprender sobre as limitações presentes nas previsões em séries temporais. Um evento atípico como a fala de uma celebridade nas redes sociais pode fazer com que o valor das ações caia abruptamente. Ao mesmo tempo, a existência de uma previsão de alta, que pode ser consultada, pode fazer com que o preço das ações suba além do previsto devido aos interesses dos investidores nesse aumento.

9.1 Consultando o preço de ações

Para que seja possível coletar o preço das ações, vamos instalar a biblioteca **yfinance** no Google Colab através do comando:

```
!pip install yfinance
```

Além da yfinance, vamos importar novamente as bibliotecas Pandas e Matplotlib, já que estamos trabalhando em um novo projeto.

```
import yfinance as yf
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

Podemos escolher o nome da ação de interesse. Aqui vamos escolher como exemplo a **TAEE11**. A sigla da ação deve ser seguida de **.SA**. Caso você deseje testar ações de outras empresas é possível consultar o nome das ações listadas na bolsa em: https://www.b3.com.br/pt_br/produtos-e-servicos/negociacao/renda-variavel/empresas-listadas.htm.

Também é possível escolher o período de tempo em que será realizada a análise através dos parâmetros `start` e `end`. Outra opção é utilizar a opção `period='max'`, que nos retorna todo o período de tempo em que existem dados.

```
df_acao = yf.download('TAEE11.SA',end='2022-02-04')
```

Como resultado, obtivemos um DataFrame com a coluna `Date` das datas e as colunas `Open`, `High`, `Low`, `Close`, `Adj Close`, que estão relacionadas respectivamente aos preços de abertura, mais alto, mais baixo e de fechamento na bolsa de valores. Por fim, a coluna `Volume` nos informa o número de transações para aquela data específica. Cada uma dessas informações pode ser expressa como uma série temporal.

```
df_acao\
  .round(2)\
  .head()
```

Date	Open	High	Low	Close	Adj Close	Volume
2007-11-13 00:00:00	11.33	11.5	11.33	11.48	3.34	33000
2007-11-14 00:00:00	11.67	11.68	11.65	11.67	3.39	100500

Date	Open	High	Low	Close	Adj Close	Volume
2007-11-16 00:00:00	11.77	11.77	11.1	11.17	3.25	41700
2007-11-19 00:00:00	11.33	11.33	11.3	11.3	3.29	7200
2007-11-21 00:00:00	11.3	11.3	10.17	10.17	2.96	16800

Como exemplo, vamos analisar a coluna `Adj Close`. Essa coluna mostra o preço de fechamento após alguns ajustes. Na sequência, vamos fazer o processo de divisão dos dados em treino e teste, a previsão com o Prophet e a construção do gráfico para que possamos verificar os resultados.

```
#Dados de treino
df_treino = pd.DataFrame()
df_treino['ds'] = df_acao.index[:3400]
df_treino['y'] = df_acao['Adj Close'][:3400].values
#Dados de teste
df_teste = pd.DataFrame()
df_teste['ds'] = df_acao.index[3400:]
df_teste['y'] = df_acao['Adj Close'][3400:].values

#Previsão com o Prophet
from fbprophet import Prophet
modelo =
Prophet(seasonality_mode='additive',daily_seasonality=False,weekly_seasona
lity=True,yearly_seasonality=True,
                 n_changepoints = 5)
modelo.fit(df_treino)
dataframefuturo = modelo.make_future_dataframe(freq='D',periods=184)
```

```
previsao = modelo.predict(dataframefuturo);

#Construção do gráfico
fig = modelo.plot(previsao)
plt.plot(df_teste['ds'],df_teste['y'],'.r')
```

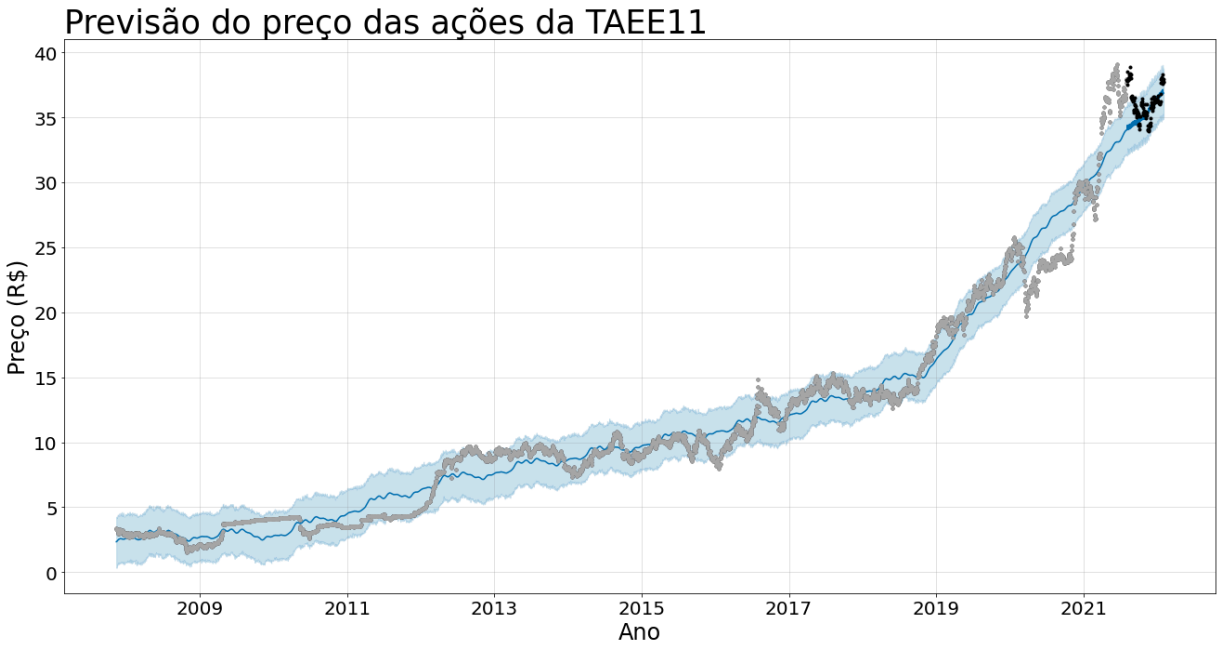

Figura 9.1: Previsão preliminar dos dados da TAEE11.

Caso você tente verificar o tamanho dos dados de teste e comparar com o tamanho do argumento `periods`, deve reparar que o valor utilizado para esse argumento é maior que o tamanho dos dados de teste. Isso acontece porque os dados que estamos utilizando não foram coletados de forma regular. Ou seja, as datas dos dados de treino possuem lacunas. Isso faz com que o Prophet tente reproduzir o padrão das datas presentes na construção do `dataframefuturo`. Desta forma, aqui foi utilizado um valor para `periods` que fizesse a última data do `dataframefuturo` coincidir com a última data dos dados de teste.

```
df_teste['ds'][125]

Output:
Timestamp('2022-02-03 00:00:00')
```

```
dataframefuturo['ds'][3583]

Output:
 Timestamp('2022-02-03 00:00:00')
```

Embora o resultado da previsão não seja perfeito, podemos considerá-lo satisfatório para o nosso objetivo de visualizar um primeiro resultado da previsão. Como argumentado no início do capítulo, vamos tentar inserir informações do dia a dia como forma de melhorar o ajuste do modelo aos dados.

9.2 Feriados

Um feriado pode ser uma data significativa se estivermos analisando a série temporal das vendas de uma loja de varejo. Os consumidores podem ter mais disposição para ir até uma loja de um setor específico. Já a bolsa de valores não opera em feriados. Porém, será que a simples presença de feriados na semana afeta de alguma forma o preço das ações? Para avaliar cada um desses casos precisamos de uma forma de adicionar essa informação ao modelo produzido pelo Prophet.

O Prophet já possui uma forma de adicionar os feriados do país que está relacionado ao nosso estudo. Nesse caso, precisamos criar o modelo e adicionar os feriados do Brasil com o método `add_country_holidays` e a propriedade `country_name='BR'` . Depois disso realizamos o processo de `fit` e construção do gráfico.

```
modelo =
Prophet(seasonality_mode='additive',daily_seasonality=False,weekly_seasona
lity=True,yearly_seasonality=True,
                 n_changepoints = 5)
#Adicionando feriados do Brasil
modelo.add_country_holidays(country_name='BR')
```

```
modelo.fit(df_treino)
dataframefuturo = modelo.make_future_dataframe(freq='D',periods=185)
previsao = modelo.predict(dataframefuturo);

#Construção do gráfico
fig = modelo.plot(previsao)
plt.plot(df_teste['ds'],df_teste['y'],'.r')
```

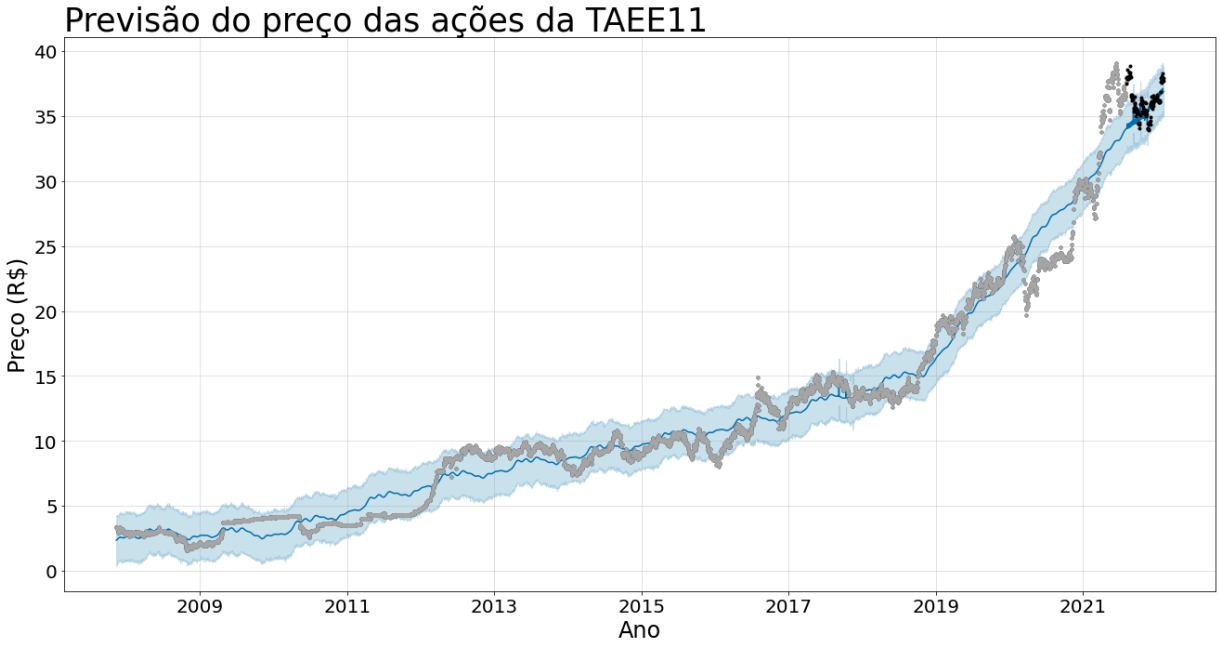

Figura 9.2: Previsão dos dados da TAEE11 adicionando os feriados nacionais.

À primeira vista, parece que o resultado não mudou muito. Você pode verificar a qualidade do ajuste e da previsão através de alguma métrica assim como fizemos nos capítulos anteriores. Mas o que é interessante verificarmos nesse momento é como os feriados estão contribuindo com o modelo. Para isso, vamos plotar as componentes:

```
modelo.plot_components(previsao)
```

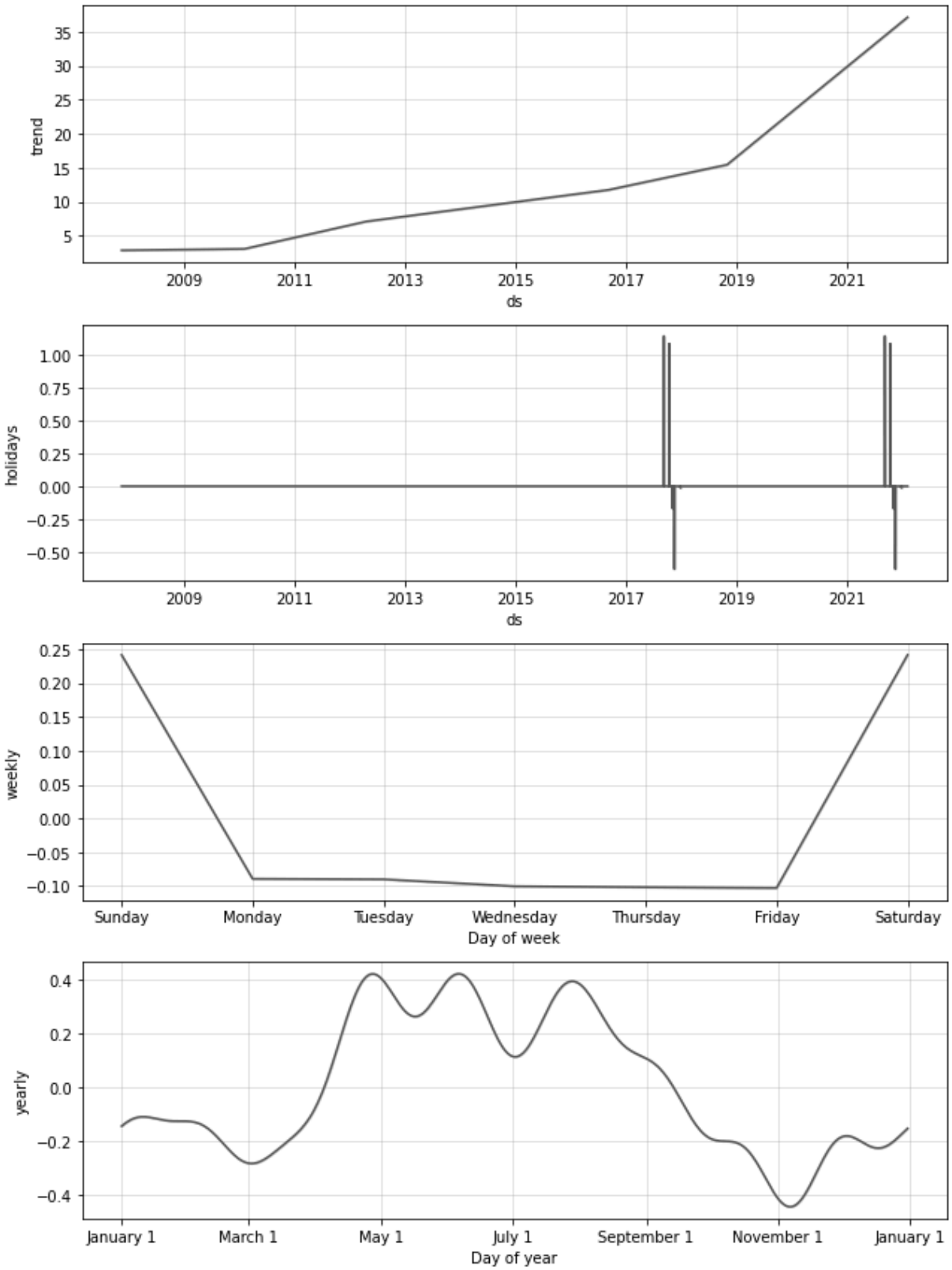

Figura 9.3: Gráficos de componentes que inclue os feriados nacionais.

Focando no gráfico de feriados (*holidays*) temos entradas pontuais que foram adicionadas com pesos diferentes. Alguns dos feriados têm uma amplitude maior que outros. Além disso, alguns aparecem como contribuição negativa no valor das ações e outros como contribuição positiva. Mas quais feriados foram adicionados?

A opção `add_country_holidays` adiciona somente feriados nacionais. Para ter certeza de quais feriados foram adicionados temos a opção de utilizar o método `.train_holiday_names`.

```
modelo.train_holiday_names
```

```
Output:
0                 New Year's Day
1                     Tiradentes
2                   Worker's Day
3               Independence Day
4    Our Lady of the Apparition
5                 All Souls' Day
6     Republic Proclamation Day
7                      Christmas
```

Se consultarmos o nome das colunas no DataFrame de previsão, vemos que cada feriado adicionado representa uma nova coluna.

```
previsao.columns
```

```
Saída:
Index(['ds', 'trend', 'yhat_lower', 'yhat_upper', 'trend_lower',
'trend_upper',
       'All Souls' Day', 'All Souls' Day_lower', 'All Souls' Day_upper',
       'Christmas', 'Christmas_lower', 'Christmas_upper', 'Independence
Day',
       'Independence Day_lower', 'Independence Day_upper', 'New Year's
Day',
       'New Year's Day_lower', 'New Year's Day_upper',
       'Our Lady of the Apparition', 'Our Lady of the Apparition_lower',
       'Our Lady of the Apparition_upper', 'Republic Proclamation Day',
       'Republic Proclamation Day_lower', 'Republic Proclamation
```

```
Day_upper',
       'Tiradentes', 'Tiradentes_lower', 'Tiradentes_upper', 'Worker's
Day',
       'Worker's Day_lower', 'Worker's Day_upper', 'additive_terms',
       'additive_terms_lower', 'additive_terms_upper', 'holidays',
       'holidays_lower', 'holidays_upper', 'weekly', 'weekly_lower',
       'weekly_upper', 'yearly', 'yearly_lower', 'yearly_upper',
       'multiplicative_terms', 'multiplicative_terms_lower',
       'multiplicative_terms_upper', 'yhat'],
      dtype='object')
```

Para verificarmos em detalhe qual a contribuição de cada feriado em um determinado momento da série temporal podemos construir um DataFrame que contém eventos diferentes de zero com as suas respectivas datas.

Para fazer isso, pegamos as colunas de interesse no Dataframe de previsão e mudamos o nome das colunas para o português. Depois fazemos uma `query` para buscar por todas as linhas das colunas de feriados que continham valores diferentes de zero. Por fim, criamos o DataFrame `resumo_feriados` com a coluna de `ds` e a coluna de `feriados`, que contém a contribuição de cada feriado.

```
col_feriados = previsao[['ds',*modelo.train_holiday_names.tolist()]]

#renomeando as colunas
col_feriados.columns =
['ds','Ano_novo','Tiradentes','Trabalho','Independencia','Aparecida','Fina
dos','Republica','Natal']

#busca por valores diferentes de zero
contrib_feriados = col_feriados.query('Ano_novo !=0 or Tiradentes !=0 or
Trabalho != 0 or Independencia != 0 or Aparecida != 0 or Finados != 0 or
Republica != 0 or Natal != 0')

#construção do DataFrame
resumo_feriados = pd.DataFrame()
resumo_feriados['ds'] = contrib_feriados['ds']
resumo_feriados['feriados'] =
```

```
contrib_feriados[['Ano_novo','Tiradentes','Trabalho','Independencia','Aparecida','Finados','Republica','Natal']].sum(axis=1)
resumo_feriados.round(2)
```

index	ds	feriados
2431	2017-09-07 00:00:00	1.09
2456	2017-10-12 00:00:00	1.03
2471	2017-11-02 00:00:00	-0.21
2480	2017-11-15 00:00:00	-0.68
2508	2017-12-25 00:00:00	-0.07
3434	2021-09-07 00:00:00	1.09
3469	2021-10-12 00:00:00	1.03
3490	2021-11-02 00:00:00	-0.21
3503	2021-11-15 00:00:00	-0.68
3543	2021-12-25 00:00:00	-0.07

Com isso, fica possível avaliar o peso que cada feriado teve no ajuste da série temporal. Se você está trabalhando com vendas, talvez o Natal seja uma data com contribuição positiva e assim é possível avaliar se existe coerência nos resultados.

Assim como a sazonalidade os feriados contam com um parâmetro para trabalhar com o peso que os feriados têm no ajuste. O `holidays_prior_scale` tem por padrão o valor igual a 10. Se esse valor for reduzido, a importância dos feriados também será reduzida, diminuindo a amplitude dos valores ajustados à série temporal. Tente reproduzir o que fizemos até o momento com um valor igual `0.05`, por exemplo.

9.3 Datas customizadas

Nossa primeira tentativa foi utilizar datas pré-implementadas para os feriados nacionais. O problema é que podemos querer incluir os feriados municipais ou mesmo datas que não estejam em nenhum outro lugar, como uma data onde um produto específico foi lançado.

Como estamos lidando com dados de ações, vamos resgatar datas que são interessantes para os acionistas, as datas de pagamentos de dividendos. Vale ressaltar que o pagamento de dividendos pode ter uma influência mais ampla do que um evento pontual.

Por exemplo, esta matéria da Valor investe explica como o valor das ações é reduzido após determinada data do anúncio de pagamento de dividendos: https://valorinveste.globo.com/blogs/andre-rocha/post/2019/06/por-que-o-preco-das-acoes-fica-ex-apos-dividendos-e-outras-duvidas.ghtml.

Outros exemplos são os artigos *Dividends and stock prices* (https://www.jstor.org/stable/1818565) e *Effects of dividends on stock prices in Nepal* (https://www.nrb.org.np/contents/uploads/2019/12/NRB_Economic_Review-Vol_24-2_October_20125_Effects_of_Dividends_on_Stock_Prices_in_NepalRabindra_Joshi.pdf), que discutem o efeito dos dividendos no preço das ações em regiões específicas do globo.

Entretanto, nosso objetivo é utilizar as datas dos pagamentos de dividendos da mesma forma como tratamos os feriados, ou seja, simulando eventos pontuais que ocorrem em datas diferentes dos feriados nacionais. Para coletar as datas de pagamentos de dividendos da TAEE11 podemos utilizar o método `Ticker`, que vai coletar diversas informações além dos dividendos. Para ver quais outras informações estão disponíveis sobre a ação, podemos percorrer a opção `.info.items()` com um laço `for`, imprimindo o nome da opção e o valor para a ação de interesse.

```
informacoes = yf.Ticker('TAEE11.SA')
for chave, valor in informacoes.info.items():
  print(chave,':',valor)
```

Como a saída é longa, vamos omiti-la. Porém, através do método utilizado, você pode obter informações de interesse para um investidor como o *ebitda* ou as margens de lucro da empresa. Já para os dividendos, o caminho é mais direto. Utilizamos o método `.dividends` nas informações coletadas.

```
dividendos = pd.DataFrame(informacoes.dividends)
dividendos.head()
```

Date	Dividends
2008-07-31 00:00:00	0.321393
2009-03-11 00:00:00	0.034538332
2009-10-19 00:00:00	1.14784
2010-11-09 00:00:00	0.40353334
2011-04-28 00:00:00	1.26534

```
dividendos.shape
```

Saída:

```
(41, 1)
```

Com isso, podemos ver que os dividendos ocorrem em datas específicas assim como os feriados. O DataFrame de dividendos contém 41 datas onde os dividendos foram pagos. Agora vamos adicionar essas informações à previsão. Para isso, vamos adicionar as datas de pagamentos de dividendos em um DataFrame de nome `datas_dividendos`.

Repare que na construção do DataFrame também estamos incluindo os parâmetros `lower_window` e `upper_window` valendo zero. Esses parâmetros não são obrigatórios, mas é neles que você tem que

trabalhar caso considere que seu evento especial influencia mais dias. Por exemplo, se um dia anterior ao seu feriado também possui influência, o `lower_windown` deve ser igual a `-1`. Já se dois dias posteriores ao seu feriado também influenciam a série temporal, o `upper_window` deve ser igual a `2`. Que tal explorar valores diferentes?

```
datas_dividendos = pd.DataFrame({
    'holiday': 'dividendos',
    'ds': pd.to_datetime(dividendos.index),
    'lower_window': 0,
    'upper_window': 0,
})
```

Após criar o DataFrame, basta adicioná-lo na criação do modelo através do parâmetro `holidays` e realizar o ajuste do modelo.

```
modelo =
Prophet(seasonality_mode='additive',daily_seasonality=False,weekly_seasona
lity=True,yearly_seasonality=True,
                n_changepoints = 5,holidays=datas_dividendos)

modelo.fit(df_treino)
dataframefuturo = modelo.make_future_dataframe(freq='D',periods=185)
previsao = modelo.predict(dataframefuturo);
```

Já adiantando que o resultado da previsão não será muito diferente visualmente, vamos seguir para a construção das componentes. É possível comparar esse resultado com as componentes de quando adicionamos os feriados nacionais. Em relação aos feriados, o Prophet percebeu que os dividendos possuem um impacto muito menor na série temporal. Repare na escala mostrada no eixo y. Podemos cogitar que isso se deve à correção dos valores dos dividendos nos valores da cotação das ações, por exemplo.

Será que se você utilizar os dados de fechamento "Close" para a previsão o resultado será o mesmo? Vale a pena tentar.

```
modelo.plot_components(previsao,'holidays');
```

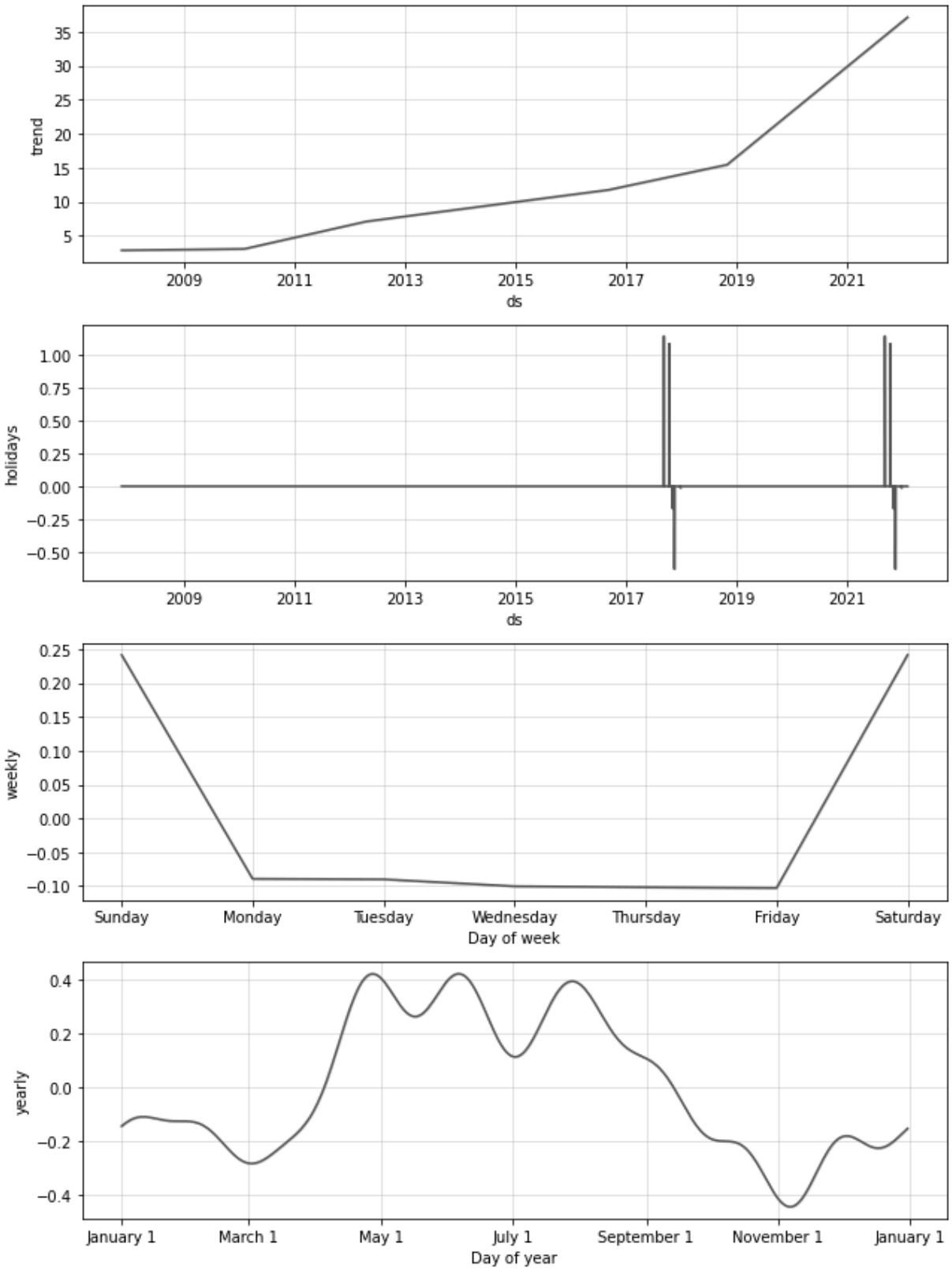

Figura 9.4: Gráficos de componentes que inclue as de pagamentos de dividendos.

9.4 Importância individual dos eventos

Para os feriados, também temos a opção `prior_scale` que podemos inserir através do parâmetro `holidays_prior_scale`. O valor padrão desse parâmetro é igual a 10 para todos os eventos. Se você diminuir esse valor, você diminuirá o efeito dos feriados ou demais datas adicionadas.

O único detalhe é que as datas ou feriados podem ter graus de importância diferentes. O Natal pode ser muito mais importante do que o feriado de Tiradentes para as suas vendas. Uma forma de lidar com esse problema é, em vez de adicionar um único valor de `prior_scale`, adicionar uma coluna de nome `prior_scale` no DataFrame de feriados.

Como exemplo, vamos criar um `array` com a biblioteca `numpy` com valores inteiros aleatórios de zero até dez e usar esse `array` na coluna `prior_scale` do `datas_dividendos`. Repare que esses valores não precisam ser inteiros e muito menos devem ser aleatórios. Você pode escolher os valores conforme a importância que a data possui no seu estudo de caso.

```
from numpy.random import randint
prior_scale = randint(0,10,41)

datas_dividendos = pd.DataFrame({
    'holiday': 'dividendos',
    'ds': pd.to_datetime(dividendos.index),
    'lower_window': 0,
    'upper_window': 0,
    'prior_scale': prior_scale
})
```

```
datas_dividendos.head()
```

index	holiday	ds	lower _window	upper _window	prio _scal
0	dividendos	2008-07-31 00:00:00	0	0	8
1	dividendos	2009-03-11 00:00:00	0	0	7
2	dividendos	2009-10-19 00:00:00	0	0	8
3	dividendos	2010-11-09 00:00:00	0	0	5
4	dividendos	2011-04-28 00:00:00	0	0	1

9.5 Gráficos de velas

Por fim, vamos plotar as nossas séries temporais usando uma forma comum do mercado de ações. Os gráficos de vela (*candlestick*) utilizam os valores das cotações das ações na abertura e fechamento do mercado e também consideram os valores de alta e de baixa de uma determinada data. Ou seja, para tentar gerar um gráfico desse

tipo vamos precisar utilizar o primeiro `DataFrame` deste capítulo. Vamos utilizar a biblioteca Plotly diretamente passando cada coluna de interesse do `DataFrame` separadamente.

```
import plotly.graph_objects as go
fig = go.Figure(data=[go.Candlestick(x=df_acao.index,
                open=df_acao['Open'],
                high=df_acao['High'],
                low=df_acao['Low'],
                close=df_acao['Close'],
                increasing_line_color= 'black', decreasing_line_color=
'gray'
)])
fig.update_layout(xaxis_rangeslider_visible=False)
fig.show()
```

Figura 9.5: Gráficos de velas contendo os preços de abertura, fechamento, alta e baixa das ações da TAEE11 .

O gráfico de saída parece com uma linha estilizada, não tão diferente dos gráficos que geramos até agora. Porém, se você clicar na área do gráfico, segurar o botão do mouse e arrastar o cursor criando uma janela, você dará zoom na área do gráfico. Com isso, podemos ver que cada "vela" que forma o gráfico. Se você passar o

mouse sobre uma delas, visualizará os valores da cotação daquele dia.

Normalmente, essas velas são mostradas nas cores verde e vermelha, mas também é possível utilizar os parâmetros `increasing_line_color` e `decreasing_line_color` para escolher outras cores. Aqui escolhemos mostrar na cor preta as velas onde o preço de fechamento da ação é maior do que o preço de abertura, ou seja, nas datas em que houve aumento de preço. Já para o caso onde houve uma diminuição de preço, estamos mostrando a cor em cinza claro.

Com isso, conseguimos entender melhor em que dias houve aumento ou diminuição de preço de uma ação. O preço de máximo e mínimo mostrados como os extremos da barra vertical nos passam a informação do tamanho da variação naquele dia. Embora cada um desses preços tenha uma relação e seja interessante visualizá-los em conjunto no Prophet, podemos tratá-los como séries temporais independentes. Assim, é possível criar um modelo para cada um dos preços.

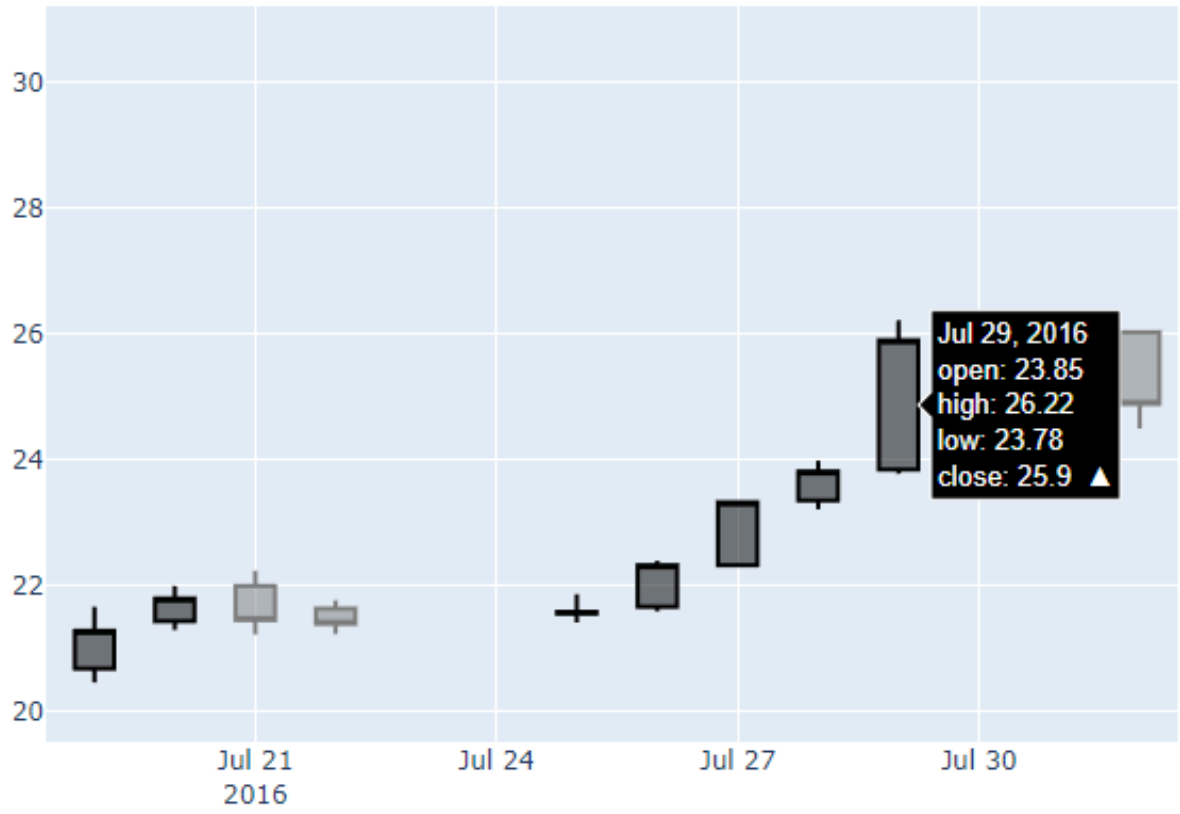

Figura 9.6: Gráficos de velas uma vela onde houve aumento de preço no dia.

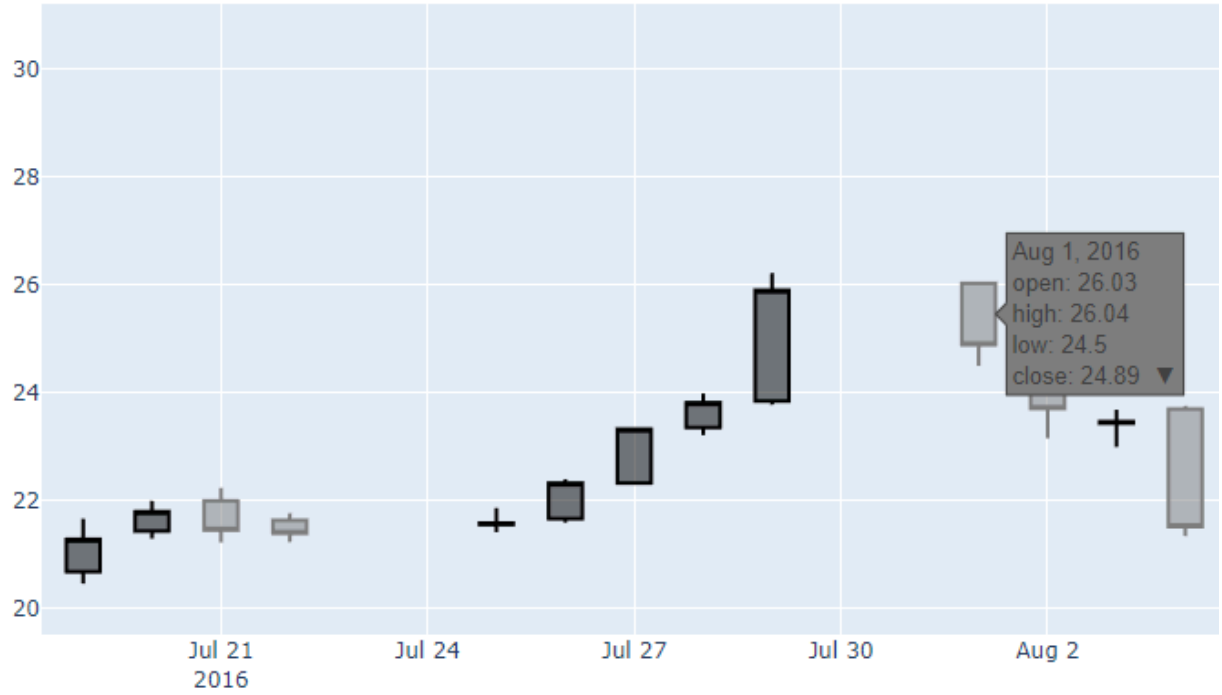

Figura 9.7: Gráficos de velas uma vela onde houve diminuição de preço no dia.

Mas e quando estamos estudando uma variável que depende de mais fatores do que apenas o tempo? É disso que vamos tratar no capítulo seguinte.

Capítulo 10
Múltiplos regressores

No decorrer do livro utilizamos o Prophet para resolver problemas onde o eixo x de um gráfico representava a variável de tempo. Nesse caso, o tempo é considerado uma variável independente. As variáveis de interesse como a temperatura, preço de um produto ou valor de uma ação representam variáveis dependentes. Porém, existem problemas onde uma variável pode depender de mais fatores e não apenas do tempo. Neste capítulo, vamos analisar um problema desse tipo.

10.1 Novos dados de temperatura

Para analisar uma forma de lidar com problemas multivariados, vamos utilizar um conjunto de dados disponível no Kaggle relacionado à variação de temperatura diária na cidade de Deli. Você pode encontrar essa base de dados no link https://www.kaggle.com/datasets/sumanthvrao/daily-climate-time-series-data. Também deixamos os arquivos disponíveis em um repositório do Github para facilitar a leitura e vamos utilizar os links na sequência de código para que possamos visualizar o DataFrame.

```
url1 = 'https://github.com/allanspadini/livro-val-
allan/raw/main/DailyDelhiClimateTrain.csv'
url2 = 'https://github.com/allanspadini/livro-val-
allan/raw/main/DailyDelhiClimateTest.csv'

import pandas as pd
dado = pd.read_csv(url1)
dado.columns = ['data','temperatura_m','umidade','vel_vento','pressao_m']
dado['data'] = pd.to_datetime(dado['data'])
dado.head()
```

index	data	temperatura_m	umidade	vel_vento	pres_r
0	2013-01-01 00:00:00	10.0	84.0	0.0	1016
1	2013-01-02 00:00:00	7.0	92.0	3.0	1018
2	2013-01-03 00:00:00	7.0	87.0	5.0	1019
3	2013-01-04 00:00:00	9.0	71.0	1.0	1017
4	2013-01-05 00:00:00	6.0	87.0	4.0	1016

Carregamos a base de dados e mudamos os nomes das colunas para uma versão em português. Nesse exemplo, temos a data que representa a nossa variável de tempo, mas também a temperatura média, umidade, velocidade do vento e pressão média. Vamos assumir que nosso objetivo é estimar um valor de temperatura média nos dias futuros. Aqui, podemos proceder como nos capítulos anteriores. Porém, será que a temperatura depende também de fatores como a velocidade do vento?

Para visualizar como a temperatura varia em relação às datas e umidade vamos utilizar a biblioteca Plotly para construir um gráfico 3D com cada ponto do DataFrame. Para este gráfico, vamos passar a umidade para o eixo x, as datas para o y e a temperatura média para o eixo z. Além disso, vamos mostrar os valores de pressão

através de uma escala de cores em cinza e os valores de velocidade do vento através da variação do tamanho dos pontos no gráfico.

```
import plotly.express as px

fig = px.scatter_3d(dado, x='umidade', y='data', z='temperatura_m',

color='pressao_m',color_continuous_scale='greys',size='vel_vento')
fig.show()
```

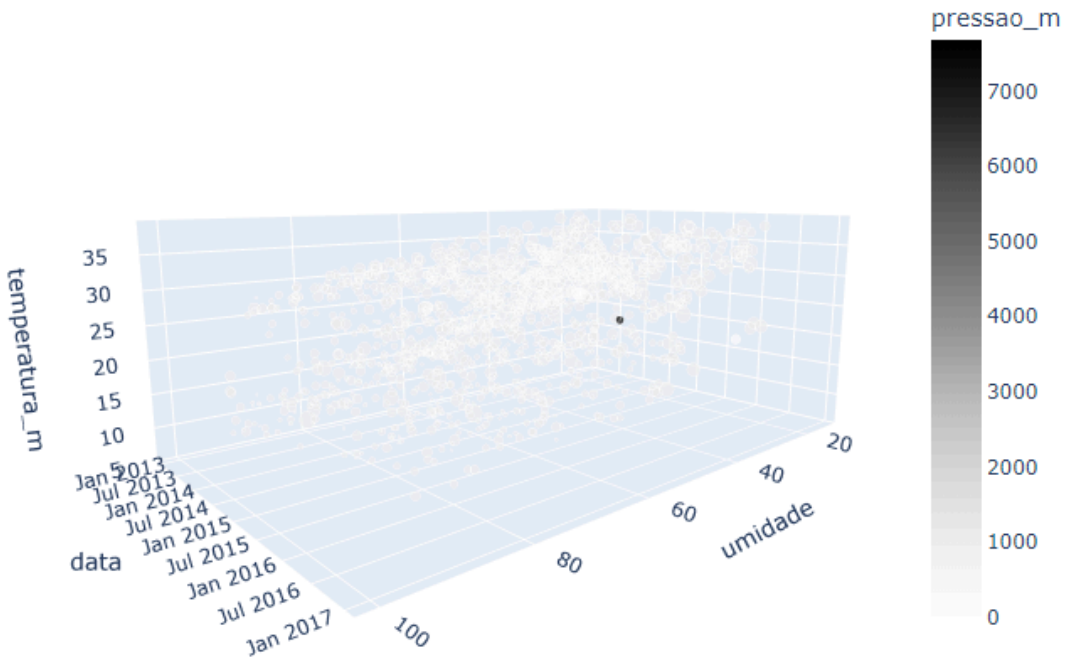

Figura 10.1: Gráfico 3D dos valores de temperatura pela umidade e tempo.

Se você reproduzir os comandos na sua máquina, poderá interagir com a figura. Clicando e segurando na figura, você consegue rotacionar a imagem. Um ponto de atenção aqui é que a maior parte dos pontos está em cor próxima da cor branca e podemos ver um único escuro na parte central da figura. Como estamos mostrando a pressão através de uma escala de cores, isso nos indica que existe um valor muito mais alto de pressão que os demais valores. Vamos

considerar este ponto um *outlier* e removê-lo usando uma `query` da biblioteca Pandas.

```
dado = dado.query('pressao_m < 7000')
```

Com isso, conseguimos gerar a figura novamente.

```
fig = px.scatter_3d(dado, x='umidade', y='data', z='temperatura_m',

color='pressao_m',color_continuous_scale='greys',size='vel_vento')
fig.show()
```

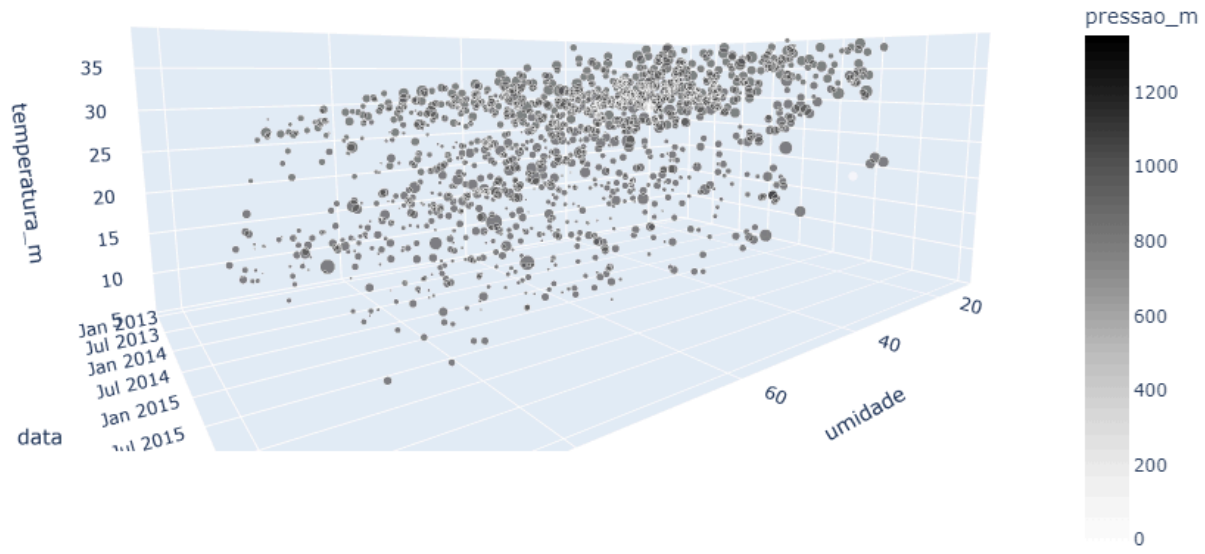

Figura 10.2: Gráfico 3D dos valores de temperatura pela umidade e tempo após a remoção do outlier.

Em relação à umidade, parece haver uma queda nos valores de temperatura conforme a umidade aumenta. Já em relação à pressão, a maior parte dos valores parece estar em uma escala muito parecida de cores tornando a interpretação difícil. O mesmo ocorre para a velocidade do vento, a variação no tamanho dos pontos parece ser aleatória.

A observação mais importante que podemos extrair desse problema é que quanto maior o número de variáveis mais difícil a interpretação. Se houvesse mais variáveis, não conseguiríamos mostrar todas ao mesmo tempo no gráfico. Existem soluções para

isso, como a visualização de cada coluna separadamente, mas deixamos isso como desafio para você.

Aqui, vamos utilizar uma estratégia baseada na correlação entre variáveis. Queremos verificar o nível de correlação das variáveis com a temperatura. Vamos calcular a correlação usando o `.corr()` e posteriormente obter o valor absoluto com o comando `abs`. Assim usamos o `imshow` da biblioteca Plotly para visualizar essas informações.

```
fig = px.imshow(abs(dado.iloc[:,
1:].corr()),color_continuous_scale='greys')
fig.show()
```

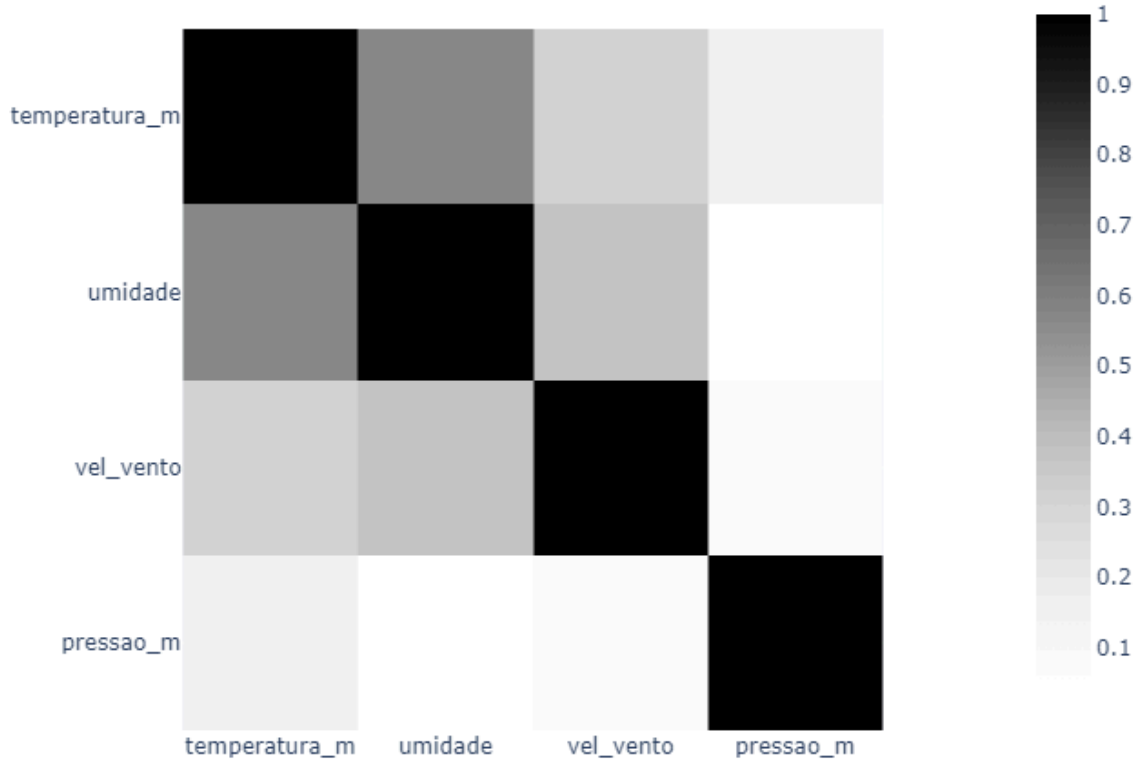

Figura 10.3: Matriz de correlação das variáveis do conjunto de dados estudado

Na figura gerada, temos diversos quadrados. Cada quadrado mostra um valor de correlação entre um par de variáveis. Nos quadrados pretos temos um valor de correlação máxima. Isso ocorre porque

estamos comparando uma variável consigo mesma. No caso entre duas variáveis distintas, quanto mais escuro o quadrado, maior a correlação. Se olharmos para a linha do topo vamos ver que, quando comparamos a temperatura com a umidade, temos o cinza mais escuro. Ou seja, a umidade é a variável que mais influencia o valor da temperatura.

Medidas de correlação sempre são uma boa opção para verificarmos quais as variáveis mais interessantes para utilizarmos dentro de um algoritmo de aprendizagem de máquina como o utilizado pelo Prophet. Para efeito de comparação, vamos começar pela utilização padrão do Prophet apenas com o tempo com variável independente.

```
# Importação das bibliotecas
from fbprophet import Prophet
import matplotlib.pyplot as plt

# Separação dos dados de treino e teste
df_treino = pd.DataFrame()

df_treino = dado.rename(columns={'temperatura_m': 'y', 'data':'ds'})

df_teste = pd.read_csv(url2)
df_teste.columns = ['ds','y','umidade','vel_vento','pressao_m']
df_teste['ds'] = pd.to_datetime(df_teste['ds'])

# Ajuste do modelo
modelo = Prophet()
modelo.fit(df_treino)
dataframefuturo = modelo.make_future_dataframe(freq='D',periods=114)

previsao = modelo.predict(dataframefuturo);

# Construção do gráfico
fig = modelo.plot(previsao,figsize=(18,6))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Meses', fontsize=24)
```

```
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.plot(df_teste['ds'],df_teste['y'],'.r')
```

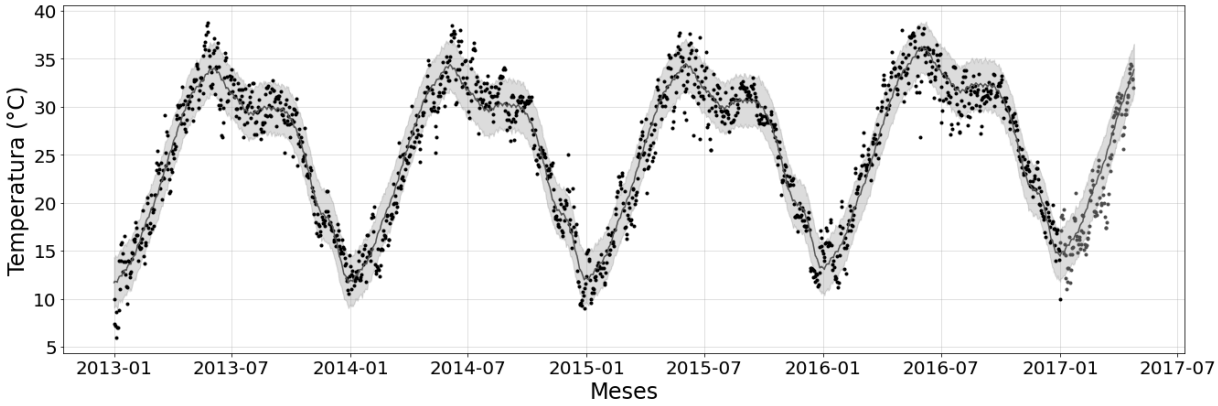

Figura 10.4: Gráfico da previsão da temperatura pelo tempo

Obtivemos um ótimo resultado. Comparando com os dados de teste, a previsão teve um bom ajuste. Como uma interpretação visual de uma melhoria neste ajuste pode ser difícil, vamos calcular o erro médio absoluto para verificar a qualidade do ajuste. Vamos calcular separadamente a qualidade para os dados de treino e de teste.

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(previsao['yhat'][:1461],df_treino['y'])
```

Output: 1.6033220602879958

```
mean_absolute_error(previsao['yhat'][1461:],df_teste['y'])
```

Output: 2.2670123361296333

Aqui vemos que o ajuste foi melhor para os dados de treino. Isso é esperado, já que os dados de treino são os únicos que o Prophet conhece.

Agora vamos testar adicionar a umidade como uma variável extra. Para fazer isso, o Prophet oferece a opção de adicionar regressores. Primeiro, instanciamos o modelo como fazemos sempre e posteriormente usamos a opção `.add_regressor`. Repare também que

nós mantivemos a coluna `umidade` nos DataFrames. Nós apenas mudamos os nomes das variáveis de tempo e temperatura.

```
modelo = Prophet()
modelo.add_regressor('umidade')
modelo.fit(df_treino)
dataframefuturo = modelo.make_future_dataframe(periods=114)
```

Até aqui fizemos o procedimento normal para a geração do `dataframefuturo`. Porém, nesse processo o Prophet não gera automaticamente uma coluna de umidade. Precisamos inserir todos os valores de umidade assim como temos as datas futuras que queremos prever. Estes valores são inseridos dentro do `dataframefuturo`. Para isso, vamos criar um DataFrame de nome `umidade` com todos os valores de treino e teste e adicionar essa coluna ao `dataframefuturo`.

```
umidade = pd.concat([df_treino['umidade'], df_teste['umidade']])
dataframefuturo['umidade'] = umidade.values
```

Com isso, conseguimos calcular a previsão da mesma forma como já estamos acostumados.

```
previsao = modelo.predict(dataframefuturo);

#Construção do gráfico
fig = modelo.plot(previsao,figsize=(18,6))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.xlabel('Meses', fontsize=24)
plt.ylabel('Temperatura (°C)', fontsize=24)
plt.plot(df_teste['ds'],df_teste['y'],'.r')
```

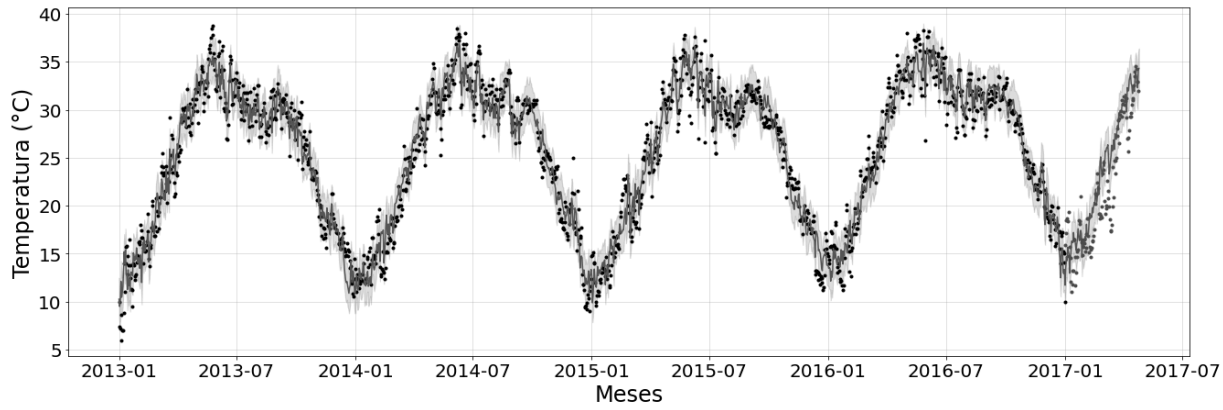

Figura 10.5: Gráfico da previsão da temperatura pelo tempo após a adição da umidade como regressor

A figura de previsão apresenta uma variabilidade muito maior quando comparada à versão em que não utilizamos o regressor extra. Isso significa que obtivemos um resultado melhor? Novamente, vamos calcular o erro médio absoluto para a previsão na parte de treino e na parte de teste separadamente.

```
mean_absolute_error(previsao['yhat'][:1461],df_treino['y'])
```

Output: 1.2684179187134377

```
mean_absolute_error(previsao['yhat'][1461:],df_teste['y'])
```

Output: 2.600312301214937

Se você comparar os resultados obtidos agora com o caso anterior, verá que obtivemos um ajuste melhor para os dados de treino no caso em que utilizamos o regressor adicional. Já para os dados de teste, a situação foi inversa. Em ambos os casos tivemos bons ajustes, mas a utilização de um regressor extra produziu sobreajuste aos dados de treino. Ou seja, o ajuste melhor aos dados de treino não se traduziu em uma melhoria ao ajuste dos dados de teste. Isso não ocorrerá em todos os casos.

Seguindo o passo a passo que fizemos, devemos escolher as variáveis com maior correlação à variável dependente e testar o ajuste tanto em dados de treino quanto em dados de teste. Além

disso, validações mais robustas como a validação cruzada são sempre importantes.

10.2 Mensagem final

Chegamos ao fim do último capítulo do livro. Espero que você tenha gostado de aprender sobre o Prophet e séries temporais.

As séries temporais são sempre desafiadoras e muitas vezes não vamos conseguir uma previsão perfeita. Porém, o Prophet é uma ferramenta intuitiva que permite um entendimento aprofundado sobre o comportamento dos dados. Existem muitas ferramentas para a análise e previsão em séries temporais. Espero que você gostado do tema e deseje se aprofundar cada vez mais nesse assunto de que gostamos tanto.